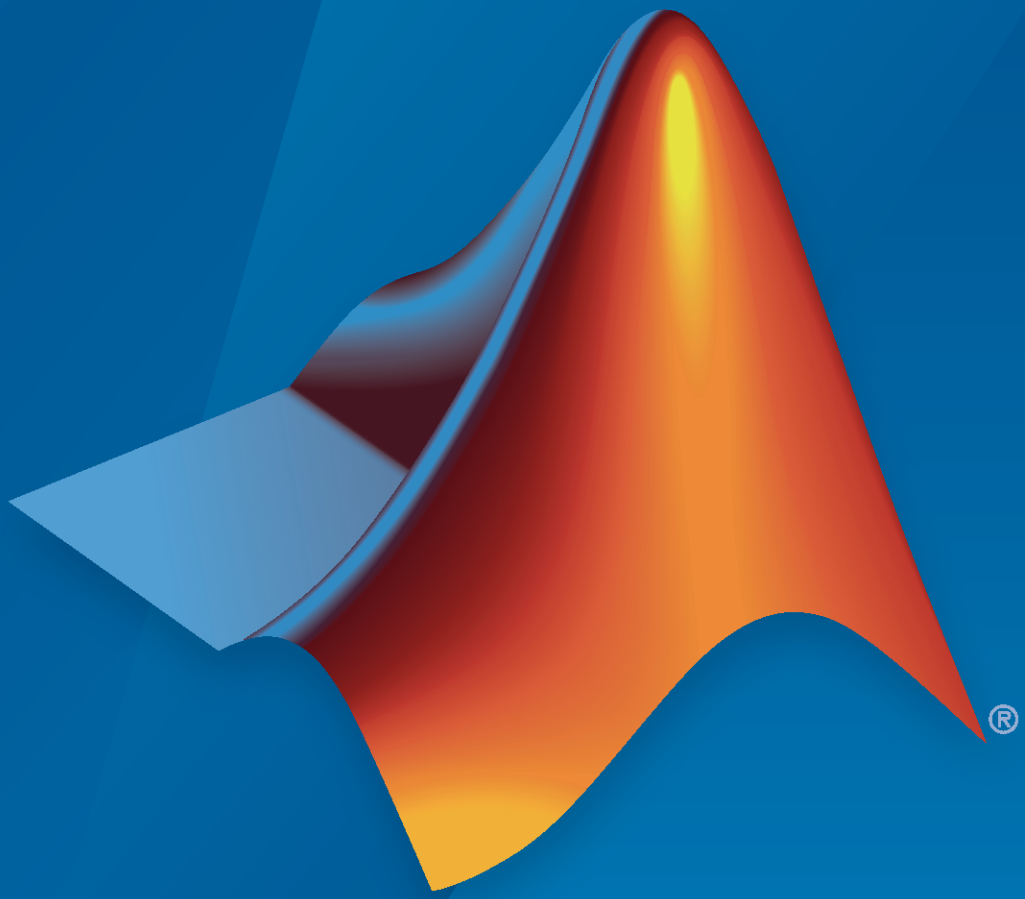
MATLAB®

Object-Oriented Programming



MATLAB®

R2023a

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Object-Oriented Programming

Revision History

March 2008	Online only	New for MATLAB 7.6 (Release 2008a)
October 2008	Online only	Revised for MATLAB 7.7 (Release 2008b)
March 2009	Online only	Revised for MATLAB 7.8 (Release 2009a)
September 2009	Online only	Revised for MATLAB 7.9 (Release 2009b)
March 2010	Online only	Revised for MATLAB 7.10 (Release 2010a)
September 2010	Online only	Revised for MATLAB 7.11 (Release 2010b)
April 2011	Online only	Revised for MATLAB 7.12 (Release 2011a)
September 2011	Online only	Revised for MATLAB 7.13 (Release 2011b)
March 2012	Online only	Revised for MATLAB 7.14 (Release 2012a)
September 2012	Online only	Revised for MATLAB 8.0 (Release 2012b)
March 2013	Online only	Revised for MATLAB 8.1 (Release 2013a)
September 2013	Online only	Revised for MATLAB 8.2 (Release 2013b)
March 2014	Online only	Revised for MATLAB 8.3 (Release 2014a)
October 2014	Online only	Revised for MATLAB 8.4 (Release 2014b)
March 2015	Online only	Revised for MATLAB 8.5 (Release 2015a)
September 2015	Online only	Revised for MATLAB 8.6 (Release 2015b)
March 2016	Online only	Revised for MATLAB 9.0 (Release 2016a)
September 2016	Online only	Revised for MATLAB 9.1 (Release 2016b)
March 2017	Online only	Revised for MATLAB 9.2 (Release 2017a)
September 2017	Online only	Revised for MATLAB 9.3 (Release 2017b)
March 2018	Online only	Revised for MATLAB 9.4 (Release 2018a)
September 2018	Online only	Revised for MATLAB 9.5 (Release 2018b)
March 2019	Online only	Revised for MATLAB 9.6 (Release 2019a)
September 2019	Online only	Revised for MATLAB 9.7 (Release 2019b)
March 2020	Online only	Revised for MATLAB 9.8 (Release 2020a)
September 2020	Online only	Revised for MATLAB 9.9 (Release 2020b)
March 2021	Online only	Revised for MATLAB 9.10 (Release 2021a)
September 2021	Online only	Revised for MATLAB 9.11 (Release 2021b)
March 2022	Online only	Revised for MATLAB 9.12 (Release 2022a)
September 2022	Online only	Revised for MATLAB 9.13 (Release 2022b)
March 2023	Online only	Revised for MATLAB 9.14 (Release 2023a)

Contents

1 Using Object-Oriented Design in MATLAB

2 Basic Example

3 MATLAB Classes Overview

6 Defining and Organizing Classes

7 Value or Handle Class — Which to Use

8 Properties — Storing Class Data

9 Methods — Defining Class Operations

10 Object Arrays

11 Events — Sending and Responding to Messages

12 How to Build on Other Classes

13 Saving and Loading Objects

14 Enumerations

15 Constant Properties

16 Information from Class Metadata

17 Specialize Object Behavior

18 Customizing Object Display

19 Defining Custom Data Types

20 Designing Related Classes

Using Object-Oriented Design in MATLAB

Why Use Object-Oriented Design

In this section...

Approaches to Writing MATLAB Programs

Creating software applications typically involves designing the application data and implementing operations performed on that data. Procedural programs pass data to functions, which perform the necessary operations on the data. Object-oriented software encapsulates data and operations in objects that interact with each other via the object's interface.

The MATLAB language enables you to create programs using both procedural and object-oriented techniques and to use objects and ordinary functions together in your programs.

Procedural Program Design

In procedural programming, your design focuses on the steps that must execute to achieve a desired state. Typically, you represent data as individual variables or fields of a structure. You implement operations as functions that take the variables as arguments. Programs usually call a sequence of functions, each one of which is passed data, and then returns modified data. Each function performs an operation or many operations on the data.

Object-Oriented Program Design

The object-oriented program design involves:

- Identifying the components of the system or application that you want to build
- Analyzing and identifying patterns to determine what components are used repeatedly or share characteristics
- Classifying components based on similarities and differences

After performing this analysis, you define classes that describe the objects your application uses.

Classes and Objects

A class describes a set of objects with common characteristics. Objects are specific instances of a class. The values contained in an object's properties are what make an object different from other objects of the same class. The functions defined by the class (called methods) are what implement object behaviors that are common to all objects of a class.

When Should You Create Object-Oriented Programs

You can implement simple programming tasks as simple functions. However, as the magnitude and complexity of your tasks increase, functions become more complex and difficult to manage.

As functions become too large, you can break them into smaller functions and pass data from one to function to another. However, as the number of functions becomes large, designing, and managing the data passed to functions becomes difficult and error prone. At this point, consider moving your MATLAB programming tasks to object-oriented designs.

Understand a Problem in Terms of Its Objects

Thinking in terms of objects is simpler and more natural for some problems. Think of the nouns in your problem statement as the objects to define and the verbs as the operations to perform.

Consider the design of classes to represent money lending institutions (banks, mortgage companies, individual money lenders, and so on). It is difficult to represent the various types of lenders as procedures. However, you can represent each one as an object that performs certain actions and contains certain data. The process of designing the objects involves identifying the characteristics of a lender that are important to your application.

Identify Commonalities

What do all money lenders have in common? All `MoneyLender` objects can have a `loan` method and an `InterestRate` property, for example.

Identify Differences

How does each money lender differ? One can provide loans to businesses while another provides loans only to individuals. Therefore, the `loan` operation might need to be different for different types of lending institutions. Subclasses of a base `MoneyLender` class can specialize the subclass versions of the `loan` method. Each lender can have a different value for its `InterestRate` property.

Factor out commonalities into a superclass and implement what is specific to each type of lender in the subclass.

Add Only What Is Necessary

These institutions might engage in activities that are not of interest to your application. During the design phase, determine what operations and data an object must contain based on your problem definition.

Objects Manage Internal State

Objects provide several useful features not available from structures and cell arrays. For example, objects can:

- Constrain the data values assigned to any given property
- Calculate the value of a property only when it is queried
- Broadcast notices when any property value is queried or changed
- Restrict access to properties and methods

Reducing Redundancy

As the complexity of your program increases, the benefits of an object-oriented design become more apparent. For example, suppose that you implement the following procedure as part of your application:

1. Check inputs
2. Perform computation on the first input argument
3. Transform the result of step 2 based on the second input argument
4. Check validity of outputs and return values

You can implement this procedure as an ordinary function. But suppose that you use this procedure again somewhere in your application, except that step 2 must perform a different computation. You

could copy and paste the first implementation, and then rewrite step 2. Or you could create a function that accepted an option indicating which computation to make, and so on. However, these options lead to more complicated code.

An object-oriented design can factor out the common code into what is called a base class. The base class would define the algorithm used and implement whatever is common to all cases that use this code. Step 2 could be defined syntactically, but not implemented, leaving the specialized implementation to the classes that you then derive from this base class.

Step 1

```
function checkInputs()
   % actual implementation
end
```

Step 2

```
function results = computeOnFirstArg()
   % specify syntax only
end
```

Step 3

```
function transformResults()
   % actual implementation
end
```

Step 4

```
function out = checkOutputs()
   % actual implementation
end
```

The code in the base class is not copied or modified. Classes you derive from the base class inherit this code. Inheritance reduces the amount of code to be tested, and isolates your program from changes to the basic procedure.

Defining Consistent Interfaces

The use of a class as the basis for similar, but more specialized classes is a useful technique in object-oriented programming. This class defines a common interface. Incorporating this kind of class into your program design enables you to:

- Identify the requirements of a particular objective
- Encode requirements into your program as an interface class

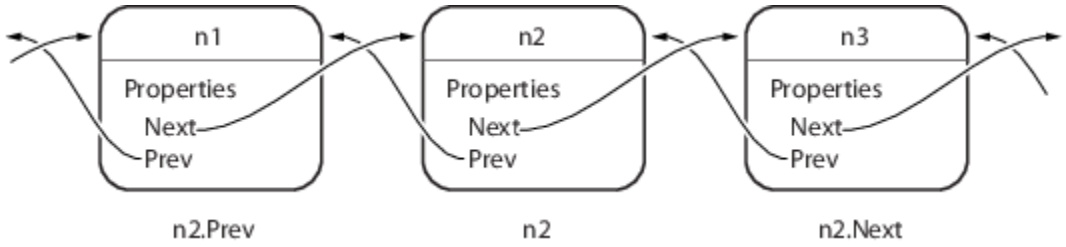
Reducing Complexity

Objects reduce complexity by reducing what you must know to use a component or system:

- Objects provide an interface that hides implementation details.
- Objects enforce rules that control how objects interact.

To illustrate these advantages, consider the implementation of a data structure called a doubly linked list. See “Implementing Linked Lists with Classes” on page 3-23 for the actual implementation.

Here is a diagram of a three-element list:



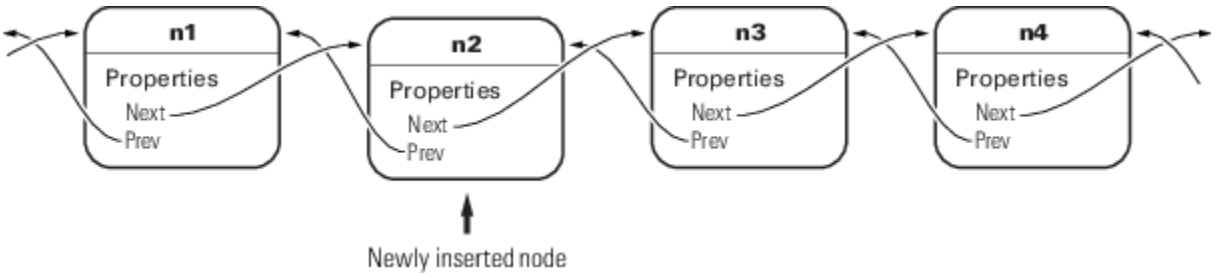
To add a node to the list, disconnect the existing nodes in the list, insert the new node, and reconnect the nodes appropriately. Here are the basic steps:

First disconnect the nodes:

1. Unlink `n2.Prev` from `n1`
2. Unlink `n1.Next` from `n2`

Now create the new node, connect it, and renumber the original nodes:

1. Link `new.Prev` to `n1`
2. Link `new.Next` to `n3` (was `n2`)
3. Link `n1.Next` to `new` (will be `n2`)
4. Link `n3.Prev` to `new` (will be `n2`)



The details of how methods perform these steps are encapsulated in the class design. Each node object contains the functionality to insert itself into or remove itself from the list.

For example, in this class, every node object has an `insertAfter` method. To add a node to a list, create the node object and then call its `insertAfter` method:

```
nnew = NodeConstructor;
nnew.insertAfter(n1)
```

Because the node class defines the code that implements these operations, this code is:

- Implemented in an optimal way by the class author
- Always up to date with the current version of the class
- Properly tested
- Can automatically update old-versions of the objects when they are loaded from MAT-files.

The object methods enforce the rules for how the nodes interact. This design removes the responsibility for enforcing rules from the applications that use the objects. It also means that the application is less likely to generate errors in its own implementation of the process.

Fostering Modularity

As you decompose a system into objects (car –> engine –> fuel system –> oxygen sensor), you form modules around natural boundaries. Classes provide three levels of control over code modularity:

- Public — Any code can access this particular property or call this method.
- Protected — Only this object's methods and methods of objects derived from this object's class can access this property or call this method.
- Private — Only the object's own methods can access this property or call this method.

Overloaded Functions and Operators

When you define a class, you can overload existing MATLAB functions to work with your new object. For example, the MATLAB serial port class overloads the `fread` function to read data from the device connected to the port represented by this object. You can define various operations, such as equality (`eq`) or addition (`plus`), for a class you have defined to represent your data.

See Also

More About

- “Role of Classes in MATLAB” on page 3-2

Handle Object Behavior

In this section...

More than one variable can refer to the same handle object. Therefore, users interact with instances of handle classes differently than instances of value classes. Understanding how handle objects behave can help you determine whether to implement a handle or a value class. This topic illustrates some of those interactions.

For more information on handle classes, see “Handle Classes”.

What Is a Handle?

Certain kinds of MATLAB objects are handles. When a variable holds a handle, it actually holds a reference to the object.

Handle objects enable more than one variable to refer to the same object. Handle-object behavior affects what happens when you copy handle objects and when you pass them to functions.

Copies of Handles

All copies of a handle object variable refer to the same underlying object. This reference behavior means that if `h` identifies a handle object, then,

```
h2 = h;
```

Creates another variable, `h2`, that refers to the same object as `h`.

For example, the MATLAB `audioplayer` function creates a handle object that contains the audio source data to reproduce a specific sound segment. The variable returned by the `audioplayer` function identifies the audio data and enables you to access object functions to play the audio.

MATLAB software includes audio data that you can load and use to create an `audioplayer` object. This sample load audio data, creates the audio player, and plays the audio:

```
load gong Fs y
gongSound = audioplayer(y,Fs);
play(gongSound)
```

Suppose that you copy the `gongSound` object handle to another variable (`gongSound2`):

```
gongSound2 = gongSound;
```

The variables `gongSound` and `gongSound2` are copies of the same handle and, therefore, refer to the same audio source. Access the `audioplayer` information using either variable.

For example, set the sample rate for the gong audio source by assigning a new value to the `SampleRate` property. First get the current sample rate and then set a new sample rate:

```
sr = gongSound.SampleRate;
disp(sr)
```

```
8192
```

```
gongSound.SampleRate = sr*2;
```

You can use `gongSound2` to access the same audio source:

```
disp(gongSound2.SampleRate)
```

```
16384
```

Play the gong sound with the new sample rate:

```
play(gongSound2)
```

Handle Objects Modified in Functions

When you pass an argument to a function, the function copies the variable from the workspace in which you call the function into the parameter variable in the function's workspace.

Passing a nonhandle variable to a function does not affect the original variable that is in the caller's workspace. For example, `myFunc` modifies a local variable called `var`, but when the function ends, the local variable `var` no longer exists:

```
function myFunc(var)
   var = var + 1;
end
```

Define a variable and pass it to `myfunc`:

```
x = 12;
myFunc(x)
```

The value of `x` has not changed after executing `myFunc(x)`:

```
disp(x)
```

```
12
```

The `myFunc` function can return the modified value, which you could assign to the same variable name (`x`) or another variable.

```
function out = myFunc(var)
   out = var + 1;
end
```

Modify a value in `myfunc`:

```
x = 12;
x = myFunc(x);
disp(x)
```

```
13
```

When the argument is a handle variable, the function copies only the handle, not the object identified by that handle. Both handles (original and local copy) refer to the same object.

When the function modifies the data referred to by the object handle, those changes are accessible from the handle variable in the calling workspace without the need to return the modified object.

For example, the `modifySampleRate` function changes the `audioplayer` sample rate:

```
function modifySampleRate(audioObj,sr)
   audioObj.SampleRate = sr;
end
```

Create an `audioplayer` object and pass it to the `modifySampleRate` function:

```
load gong Fs y
gongSound = audioplayer(y,Fs);
disp(gongSound.SampleRate)
```

```
8192
```

```
modifySampleRate(gongSound,16384)
disp(gongSound.SampleRate)
```

```
16384
```

The `modifySampleRate` function does not need to return a modified `gongSound` object because `audioplayer` objects are handle objects.

Determine If an Object Is a Handle

Handle objects are members of the `handle` class. Therefore, you can always identify an object as a handle using the `isa` function. `isa` returns logical `true` (`1`) when testing for a handle variable:

```
load gong Fs y
gongSound = audioplayer(y,Fs);
isa(gongSound,'handle')
```

To determine if a variable is a valid handle object, use `isa` and `isvalid`:

```
if isa(gongSound,'handle') && isvalid(gongSound)
   ...
end
```

Deleted Handle Objects

When a handle object has been deleted, the handle variables that referenced the object can still exist. These variables become invalid because the object they referred to no longer exists. Calling `delete` on the object removes the object, but does not clear handle variables.

For example, create an `audioplayer` object:

```
load gong Fs y
gongSound = audioplayer(y,Fs);
```

The output argument, `gongSound`, is a handle variable. Calling `delete` deletes the object along with the audio source information it contains:

```
delete(gongSound)
```

However, the handle variable still exists:

```
disp(gongSound)
```

```
handle to deleted audioplayer
```

The `whos` command shows `gongSound` as an `audioplayer` object:

```
whos
```

```
  Name               Size             Bytes  Class          Attributes

  Fs                 1x1                  8  double
  gongSound          1x1                  0  audioplayer
  y              42028x1             336224  double
```

Note The value for Bytes returned by the `whos` command does not include the data referenced by a handle because many variables can reference the same data.

The handle `gongSound` no longer refers to a valid object, as shown by the `isvalid` handle method:

```
isvalid(gongSound)
```

```
ans =

  logical

   0
```

Calling `delete` on a deleted handle does nothing and does not cause an error. You can pass an array containing both valid and invalid handles to `delete`. MATLAB deletes the valid handles, but does not issue an error when encountering handles that are already invalid.

You cannot access properties with the invalid handle variable:

```
gongSound.SampleRate
```

```
Invalid or deleted object.
```

Functions and methods that access object properties cause an error:

```
play(gongSound)
```

```
Invalid or deleted object.
```

To remove the variable, `gongSound`, use `clear`:

```
clear gongSound
whos
```

```
  Name          Size             Bytes  Class     Attributes

  Fs            1x1                  8  double
  y         42028x1             336224  double
```

See Also

More About

- “Handle Class Destructor” on page 7-13
- “Comparison of Handle and Value Classes” on page 7-2

Basic Example

Creating a Simple Class

In this section...

Design Class

The basic purpose of a class is to define an object that encapsulates data and the operations performed on that data. For example, `BasicClass` defines a property and two methods that operate on the data in that property:

- `Value` — Property that contains the numeric data stored in an object of the class
- `roundOff` — Method that rounds the value of the property to two decimal places
- `multiplyBy` — Method that multiplies the value of the property by the specified number

Start a class definition with a `classdef` *`ClassName`*`...end` block, and then define the class properties and methods inside that block. Here is the definition of `BasicClass`:

```
classdef BasicClass
   properties
      Value {mustBeNumeric}
   end
   methods
      function r = roundOff(obj)
         r = round([obj.Value],2);
      end
      function r = multiplyBy(obj,n)
         r = [obj.Value]*n;
      end
   end
end
```

For a summary of class syntax, see `classdef`.

To use the class:

- Save the class definition in a `.m` file with the same name as the class.
- Create an object of the class.
- Access the properties to assign data.
- Call methods to perform operation on the data.

Create Object

Create an object of the class using the class name:

```
a = BasicClass
```

```
a =

  BasicClass with properties:

    Value: []
```

Initially, the property value is empty.

Access Properties

Assign a value to the `Value` property using the object variable and a dot before the property name:

```
a.Value = pi/3;
```

To return a property value, use dot notation without the assignment:

```
a.Value
```

```
ans =

    1.0472
```

For information on class properties, see “Property Syntax” on page 8-4.

Call Methods

Call the `roundOff` method on object `a`:

```
roundOff(a)
```

```
ans =

    1.0500
```

Pass the object as the first argument to a method that takes multiple arguments, as in this call to the `multiplyBy` method:

```
multiplyBy(a,3)
```

```
ans =

    3.1416
```

You can also call a method using dot notation:

```
a.multiplyBy(3)
```

Passing the object as an explicit argument is not necessary when using dot notation. The notation uses the object to the left of the dot.

For information on class methods, see “Method Syntax” on page 5-7.

Add Constructor

Classes can define a special method to create objects of the class, called a constructor. Constructor methods enable you to pass arguments to the constructor, which you can assign as property values. The `BasicClass` `Value` property restricts its possible values using the `mustBeNumeric` function.

Here is a constructor for the `BasicClass` class. When you call the constructor with an input argument, it is assigned to the `Value` property. If you call the constructor without an input argument, the `Value` property has a default value of empty (`[]`).

```
methods
    function obj = BasicClass(val)
        if nargin == 1
            obj.Value = val;
        end
    end
end
```

By adding this constructor to the class definition, you can create an object and set the property value in one step:

```
a = BasicClass(pi/3)
```

```
a =

  BasicClass with properties:

    Value: 1.0472
```

The constructor can perform other operations related to creating objects of the class.

For information on constructors, see “Class Constructor Methods” on page 9-15.

Vectorize Methods

MATLAB enables you to vectorize operations. For example, you can add a number to a vector:

```
[1 2 3] + 2
```

```
ans =

     3     4     5
```

MATLAB adds the number 2 to each of the elements in the array `[1 2 3]`. To vectorize the arithmetic operator methods, enclose the `obj.Value` property reference in brackets.

```
[obj.Value] + 2
```

This syntax enables the method to work with arrays of objects. For example, create an object array using indexed assignment.

```
obj(1) = BasicClass(2.7984);
obj(2) = BasicClass(sin(pi/3));
obj(3) = BasicClass(7);
```

These two expressions are equivalent.

```
[obj.Value] + 2
[obj(1).Value obj(2).Value obj(3).Value] + 2
```

The `roundOff` method is vectorized because the property reference is enclosed in brackets.

```
r = round([obj.Value],2);
```

Because `roundOff` is vectorized, it can operate on arrays.

```
roundOff(obj)

ans =

    2.8000    0.8700    7.0000
```

Overload Functions

Classes can implement existing functionality, such as addition, by defining a method with the same name as the existing MATLAB function. For example, suppose that you want to add two `BasicClass` objects. It makes sense to add the values of the `Value` properties of each object.

Here is an overloaded version of the MATLAB `plus` function. It defines addition for the `BasicClass` class as adding the property values:

```
methods
   function r = plus(o1,o2)
      r = [o1.Value] + [o2.Value];
   end
end
```

By implementing a method called `plus`, you can use the "+" operator with objects of `BasicClass`.

```
a = BasicClass(pi/3);
b = BasicClass(pi/4);
a + b

ans =

    1.8326
```

By vectorizing the plus method, you can operate on object arrays.

```
a = BasicClass(pi/3);
b = BasicClass(pi/4);
c = BasicClass(pi/2);
ar = [a b];
ar + c

ans =

    2.6180    2.3562
```

Related Information

For information on overloading functions, see "Overload Functions in Class Definitions" on page 9-25.

For information on overloading operators, see "Operator Overloading" on page 17-19.

BasicClass Code Listing

Here is the `BasicClass` definition after adding the features discussed in this topic:

```
classdef BasicClass
    properties
        Value {mustBeNumeric}
    end
    methods
        function obj = BasicClass(val)
            if nargin == 1
                obj.Value = val;
            end
        end
        function r = roundOff(obj)
            r = round([obj.Value],2);
        end
        function r = multiplyBy(obj,n)
            r = [obj.Value] * n;
        end
        function r = plus(o1,o2)
            r = [o1.Value] + [o2.Value];
        end
    end
end
```

See Also

Related Examples

- “Components of a Class” on page 5-2
- “Validate Property Values” on page 8-18

MATLAB Classes Overview

Role of Classes in MATLAB

In this section...

Classes

In the MATLAB language, every value is assigned to a class. For example, creating a variable with an assignment statement constructs a variable of the appropriate class:

```
a = 7;
b = 'some text';
s.Name = 'Nancy';
s.Age = 64;
whos
```

```
whos
  Name      Size            Bytes  Class     Attributes

  a         1x1                 8  double
  b         1x9                18  char
  s         1x1               370  struct
```

Basic commands like `whos` display the class of each value in the workspace. This information helps MATLAB users recognize that some values are characters and display as text while other values are double precision numbers, and so on. Some variables can contain different classes of values like structures.

Predefined Classes

MATLAB defines fundamental classes that comprise the basic types used by the language. These classes include numeric, `logical`, `char`, `cell`, `struct`, and function handle.

User-Defined Classes

You can create your own MATLAB classes. For example, you could define a class to represent polynomials. This class could define the operations typically associated with MATLAB classes, like addition, subtraction, indexing, displaying in the command window, and so on. These operations would need to perform the equivalent of polynomial addition, polynomial subtraction, and so on. For example, when you add two polynomial objects:

```
p1 + p2
```

the `plus` operation must be able to add polynomial objects because the polynomial class defines this operation.

When you define a class, you can overload special MATLAB functions (such as `plus.m` for the addition operator). MATLAB calls these methods when users apply those operations to objects of your class.

See “Representing Polynomials with Classes” on page 19-2 for an example that creates just such a class.

MATLAB Classes — Key Terms

MATLAB classes use the following words to describe different parts of a class definition and related concepts.

- Class definition — Description of what is common to every instance of a class.
- Properties — Data storage for class instances
- Methods — Special functions that implement operations that are usually performed only on instances of the class
- Events — Messages defined by classes and broadcast by class instances when some specific action occurs
- Attributes — Values that modify the behavior of properties, methods, events, and classes
- Listeners — Objects that respond to a specific event by executing a callback function when the event notice is broadcast
- Objects — Instances of classes, which contain actual data values stored in the objects' properties
- Subclasses — Classes that are derived from other classes and that inherit the methods, properties, and events from those classes (subclasses facilitate the reuse of code defined in the superclass from which they are derived).
- Superclasses — Classes that are used as a basis for the creation of more specifically defined classes (that is, subclasses).
- Packages — Folders that define a scope for class and function naming

Some Basic Relationships

This section discusses some of the basic concepts used by MATLAB classes.

Classes

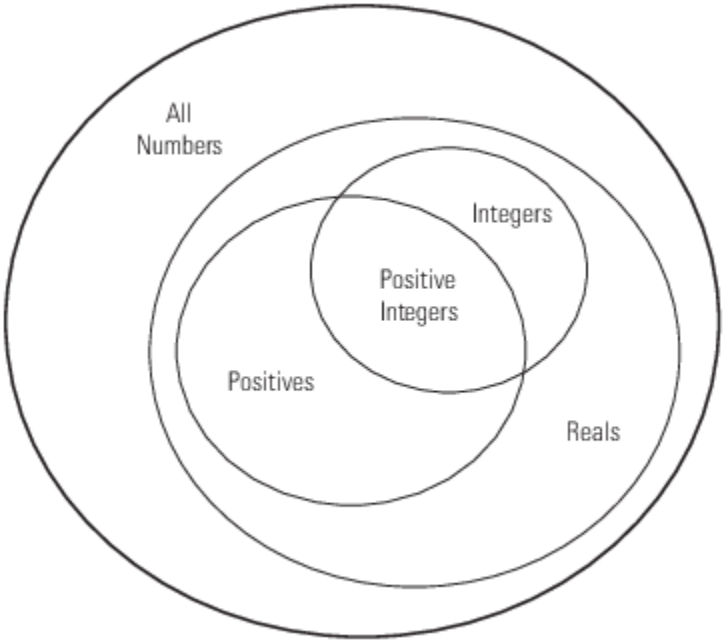
A class is a definition that specifies certain characteristics that all instances of the class share. These characteristics are determined by the properties, methods, and events that define the class and the values of attributes that modify the behavior of each of these class components. Class definitions describe how objects of the class are created and destroyed, what data the objects contain, and how you can manipulate this data.

Class Hierarchies

It sometimes makes sense to define a new class in terms of existing classes. This approach enables you to reuse the designs and techniques in a new class that represents a similar entity. You accomplish this reuse by creating a subclass. A subclass defines objects that are a subset of those objects defined by the superclass. A subclass is more specific than its superclass and might add new properties, methods, and events to those components inherited from the superclass.

Mathematical sets can help illustrate the relationships among classes. In the following diagram, the set of Positive Integers is a subset of the set of Integers and a subset of Positives. All three sets are subsets of Reals, which is a subset of All Numbers.

The definition of Positive Integers requires the additional specification that members of the set be greater than zero. Positive Integers combine the definitions from both Integers and Positives. The resulting subset is more specific, and therefore more narrowly defined, than the supersets, but still shares all the characteristics that define the supersets.



The “is a” relationship is a good way to determine if it is appropriate to define a particular subset in terms of existing supersets. For example, each of the following statements makes senses:

- A Positive Integer is an Integer
- A Positive Integer is a Positive number

If the “is a” relationship holds, then it is likely you can define a new class from a class or classes that represent some more general case.

Reusing Solutions

Classes are usually organized into taxonomies to foster code reuse. For example, if you define a class to implement an interface to the serial port of a computer, it would probably be similar to a class designed to implement an interface to the parallel port. To reuse code, you could define a superclass that contains everything that is common to the two types of ports, and then derive subclasses from the superclass in which you implement only what is unique to each specific port. Then the subclasses would inherit all the common functionality from the superclass.

Objects

A class is like a template for the creation of a specific instance of the class. This instance or object contains actual data for a particular entity that is represented by the class. For example, an instance of a bank account class is an object that represents a specific bank account, with an actual account number and an actual balance. This object has built into it the ability to perform operations defined by the class, such as making deposits to and withdrawals from the account balance.

Objects are not just passive data containers. Objects actively manage the data contained by allowing only certain operations to be performed, by hiding data that does not need to be public, and by preventing external clients from misusing data by performing operations for which the object was not designed. Objects even control what happens when they are destroyed.

Encapsulating Information

An important aspect of objects is that you can write software that accesses the information stored in the object via its properties and methods without knowing anything about how that information is stored, or even whether it is stored or calculated when queried. The object isolates code that accesses the object from the internal implementation of methods and properties. You can define classes that hide both data and operations from any methods that are not part of the class. You can then implement whatever interface is most appropriate for the intended use.

References

[1] Shalloway, A., J. R. Trott, *Design Patterns Explained A New Perspective on Object-Oriented Design*.. Boston, MA: Addison-Wesley 2002.

[2] Gamma, E., R. Helm, R. Johnson, J. Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley 1995.

[3] Freeman, E., Elisabeth Freeman, Kathy Sierra, Bert Bates, *Head First Design Patterns*. Sebastopol, CA 2004.

See Also

Related Examples

- “Creating a Simple Class” on page 2-2
- “Developing Classes That Work Together” on page 3-6
- “Representing Structured Data with Classes” on page 3-14
- “Implementing Linked Lists with Classes” on page 3-23

Developing Classes That Work Together

In this section...

Formulating a Class

This example discusses how to approach the design and implementation of two classes that interact through events and listeners. The two classes represent a bank account and an account manager.

To design a class that represents a bank account, first determine the elements of data and the operations that form your abstraction of a bank account. For example, a bank account has:

- An account number
- An account balance
- A status (open, closed, etc.)

You must perform certain operations on a bank account:

- Create an object for each bank account
- Deposit money
- Withdraw money
- Generate a statement
- Save and load the `BankAccount` object

If the balance is too low and you attempt to withdraw money, the bank account broadcasts a notice. When this event occurs, the bank account broadcasts a notice to other entities that are designed to listen for these notices. In this example, a simplified version of an account manager program performs this task.

In this example, an account manager program determines the status of all bank accounts. This program monitors the account balance and assigns one of three values:

- `open` — Account balance is a positive value
- `overdrawn` — Account balance is overdrawn, but by $200 or less.
- `closed` — Account balance is overdrawn by more than $200.

These features define the requirements of the `BankAccount` and `AccountManager` classes. Include only what functionality is required to meet your specific objectives. Support special types of accounts by subclassing `BankAccount` and adding more specific features to the subclasses. Extend the `AccountManager` as required to support new account types.

Specifying Class Components

Classes store data in properties, implement operations with methods, and support notifications with events and listeners. Here is how the `BankAccount` and `AccountManager` classes define these components.

Class Data

The class defines these properties to store the account number, account balance, and the account status:

- `AccountNumber` — A property to store the number identifying the specific account. MATLAB assigns a value to this property when you create an instance of the class. Only `BankAccount` class methods can set this property. The `SetAccess` attribute is `private`.
- `AccountBalance` — A property to store the current balance of the account. The class operation of depositing and withdrawing money assigns values to this property. Only `BankAccount` class methods can set this property. The `SetAccess` attribute is `private`.
- `AccountStatus` — The `BankAccount` class defines a default value for this property. The `AccountManager` class methods change this value whenever the value of the `AccountBalance` falls below `0`. The `Access` attribute specifies that only the `AccountManager` and `BankAccount` classes have access to this property.
- `AccountListener` — Storage for the `InsufficentFunds` event listener. Saving a `BankAccount` object does not save this property because you must recreate the listener when loading the object.

Class Operations

These methods implement the operations defined in the class formulation:

- `BankAccount` — Accepts an account number and an initial balance to create an object that represents an account.
- `deposit` — Updates the `AccountBalance` property when a deposit transaction occurs
- `withdraw` — Updates the `AccountBalance` property when a withdrawal transaction occurs
- `getStatement` — Displays information about the account
- `loadobj` — Recreates the account manager listener when you load the object from a MAT-file.

Class Events

The account manager program changes the status of bank accounts that have negative balances. To implement this action, the `BankAccount` class triggers an event when a withdrawal results in a negative balance. Therefore, the triggering of the `InsufficientsFunds` event occurs from within the `withdraw` method.

To define an event, specify a name within an `events` block. Trigger the event by a call to the `notify` handle class method. Because `InsufficientsFunds` is not a predefined event, you can name it with any `char` vector and trigger it with any action.

BankAccount Class Implementation

It is important to ensure that there is only one set of data associated with any object of a `BankAccount` class. You would not want independent copies of the object that could have, for

example, different values for the account balance. Therefore, implement the `BankAccount` class as a handle class. All copies of a given handle object refer to the same data.

BankAccount Class Synopsis

BankAccount Class	Discussion
`classdef BankAccount < handle`	Handle class because there should be only one copy of any instance of `BankAccount`. “Comparison of Handle and Value Classes” on page 7-2

`properties (Access = ?AccountManager)`
`  AccountStatus = 'open'`
`end`
	`AccountStatus` contains the status of the account determined by the current balance. Access is limited to the `BankAccount` and `AccountManager` classes. “Class Members Access” on page 12-23

`properties (SetAccess = private)`
`  AccountNumber`
`  AccountBalance`
`end`
`properties (Transient)`
`  AccountListener`
`end`
	`AccountStatus` property access by `AccountManager` class methods.
	
	`AccountNumber` and `AccountBalance` properties have private set access.
	
	`AccountListener property` is transient so the listener handle is not saved.
	
	See “Property Attributes” on page 8-8.

`events`
`  InsufficientFunds`
`end`
	Class defines event called `InsufficentFunds`. `withdraw` method triggers event when account balance becomes negative.
	
	For information on events and listeners, see “Events” .

`methods`	Block of ordinary methods. See “Method Syntax” on page 5-7 for syntax.

`  function BA = BankAccount(AccountNumber,InitialBalance)`
`  BA.AccountNumber = AccountNumber;`
`  BA.AccountBalance = InitialBalance;`
`  BA.AccountListener = AccountManager.addAccount(BA);`
`  end`
	Constructor initializes property values with input arguments.
	
	`AccountManager.addAccount` is static method of `AccountManager` class. Creates listener for `InsufficientFunds` event and stores listener handle in `AccountListener` property.

`  function deposit(BA,amt)`
`  BA.AccountBalance = BA.AccountBalance + amt;`
`  if BA.AccountBalance > 0`
`  BA.AccountStatus = 'open';`
`  end`
`  end`
	`deposit` adjusts value of `AccountBalance` property.
	
	If `AccountStatus` is `closed` and subsequent deposit brings `AccountBalance` into positive range, then `AccountStatus` is reset to `open`.

BankAccount Class	Discussion

`function withdraw(BA,amt)`
`  if (strcmp(BA.AccountStatus,'closed')&& ...`
`  BA.AccountBalance < 0)`
`  disp(['Account ',num2str(BA.AccountNumber),...`
`  ' has been closed.'])`
`  return`
`  end`
`  newbal = BA.AccountBalance - amt;`
`  BA.AccountBalance = newbal;`
`  if newbal < 0`
`  notify(BA,'InsufficientFunds')`
`  end`
`end`
	Updates `AccountBalance` property. If value of account balance is negative as a result of the withdrawal, `notify` triggers `InsufficentFunds` event.
	
	For more information on listeners, see “Events and Listeners Syntax” on page 11-18.

`function getStatement(BA)`
`  disp('--------------------------')`
`  disp(['Account: ',num2str(BA.AccountNumber)])`
`  ab = sprintf('%0.2f',BA.AccountBalance);`
`  disp(['CurrentBalance: ',ab])`
`  disp(['Account Status: ',BA.AccountStatus])`
`  disp('--------------------------')`
`end`
`end`
	Display selected information about the account. This section also ends the ordinary methods block.

`methods (Static)`	Beginning of static methods block. See “Static Methods” on page 9-23

`function obj = loadobj(s)`
`  if isstruct(s)`
`  accNum = s.AccountNumber;`
`  initBal = s.AccountBalance;`
`  obj = BankAccount(accNum,initBal);`
`  else`
`  obj.AccountListener = AccountManager.addAccount(s);`
`  end`
`end`
	`loadobj` method:
	
	• If the load operation fails, create the object from a `struct`.
	• Recreates the listener using the newly created `BankAccount` object as the source.
	
	For more information on saving and loading objects, see “Save and Load Process for Objects” on page 13-2

`end` `end`	End of static methods block End of `classdef`

Expand for Class Code

```
classdef BankAccount < handle
   properties (Access = ?AccountManager)
      AccountStatus = 'open'
   end
   properties (SetAccess = private)
      AccountNumber
      AccountBalance
   end
   properties (Transient)
      AccountListener
   end
   events
      InsufficientFunds
   end
   methods
      function BA = BankAccount(accNum,initBal)
         BA.AccountNumber = accNum;
         BA.AccountBalance = initBal;
         BA.AccountListener =  AccountManager.addAccount(BA);
      end
      function deposit(BA,amt)
```

```
            BA.AccountBalance = BA.AccountBalance + amt;
            if BA.AccountBalance > 0
               BA.AccountStatus = 'open';
            end
         end
         function withdraw(BA,amt)
            if (strcmp(BA.AccountStatus,'closed')&& BA.AccountBalance <= 0)
               disp(['Account ',num2str(BA.AccountNumber),' has been closed.'])
               return
            end
            newbal = BA.AccountBalance - amt;
            BA.AccountBalance = newbal;
            if newbal < 0
               notify(BA,'InsufficientFunds')
            end
         end
         function getStatement(BA)
            disp('-------------------------')
            disp(['Account: ',num2str(BA.AccountNumber)])
            ab = sprintf('%0.2f',BA.AccountBalance);
            disp(['CurrentBalance: ',ab])
            disp(['Account Status: ',BA.AccountStatus])
            disp('-------------------------')
         end
      end
      methods (Static)
         function obj = loadobj(s)
            if isstruct(s)
               accNum = s.AccountNumber;
               initBal = s.AccountBalance;
               obj = BankAccount(accNum,initBal);
            else
               obj.AccountListener = AccountManager.addAccount(s);
            end
         end
      end
   end
```

Formulating the AccountManager Class

The purpose of the `AccountManager` class is to provide services to accounts. For the `BankAccount` class, the `AccountManager` class listens for withdrawals that cause the balance to drop into the negative range. When the `BankAccount` object triggers the `InsufficientsFunds` event, the `AccountManager` resets the account status.

The `AccountManager` class stores no data so it does not need properties. The `BankAccount` object stores the handle of the listener object.

The `AccountManager` performs two operations:

- Assign a status to each account as a result of a withdrawal
- Adds an account to the system by monitoring account balances.

Class Components

The `AccountManager` class implements two methods:

- `assignStatus` — Method that assigns a status to a `BankAccount` object. Serves as the listener callback.
- `addAccount` — Method that creates the `InsufficientFunds` listener.

Implementing the AccountManager Class

The `AccountManager` class implements both methods as static because there is no need for an `AccountManager` object. These methods operate on `BankAccount` objects.

The `AccountManager` is not intended to be instantiated. Separating the functionality of the `AccountManager` class from the `BankAccount` class provides greater flexibility and extensibility. For example, doing so enables you to:

- Extend the `AccountManager` class to support other types of accounts while keeping the individual account classes simple and specialized.
- Change the criteria for the account status without affecting the compatibility of saved and loaded `BankAccount` objects.
- Develop an `Account` superclass that factors out what is common to all accounts without requiring each subclass to implement the account management functionality

AccountManager Class Synopsis

AccountManager Class	Discussion
`classdef AccountManager`	This class defines the `InsufficentFunds` event listener and the listener callback.
`   methods (Static)`	There is no need to create an instance of this class so the methods defined are static. See “Static Methods” on page 9-23.

`   function assignStatus(BA)`
`      if BA.AccountBalance < 0`
`         if BA.AccountBalance < -200`
`            BA.AccountStatus = 'closed';`
`         else`
`            BA.AccountStatus = 'overdrawn';`
`         end`
`      end`
`   end`
	The `assignStatus` method is the callback for the `InsufficentFunds` event listener. It determines the value of a `BankAccount` object `AccountStatus` property based on the value of the `AccountBalance` property.
	
	The `BankAccount` class constructor calls the `AccountManager` `addAccount` method to create and store this listener.

`   function lh = addAccount(BA)`
`      lh = addlistener(BA, 'InsufficientFunds', ...`
`         @(src, ~)AccountManager.assignStatus(src));`
`   end`
	`addAccount` creates the listener for the `InsufficentFunds` event that the `BankAccount` class defines.
	
	See “Control Listener Lifecycle” on page 11-23

`   end` `end`	`end` statements for `methods` and for `classdef`.

Expand for Class Code

```
classdef AccountManager
   methods (Static)
      function assignStatus(BA)
         if BA.AccountBalance < 0
            if BA.AccountBalance < -200
               BA.AccountStatus = 'closed';
            else
```

```
                BA.AccountStatus = 'overdrawn';
            end
         end
      end
      function lh = addAccount(BA)
         lh = addlistener(BA, 'InsufficientFunds', ...
            @(src, ~)AccountManager.assignStatus(src));
      end
   end
end
```

Using BankAccount Objects

The `BankAccount` class, while overly simple, demonstrates how MATLAB classes behave. For example, create a `BankAccount` object with an account number and an initial deposit of $500:

```
BA = BankAccount(1234567,500)
```

```
BA =

  BankAccount with properties:

      AccountNumber: 1234567
     AccountBalance: 500
    AccountListener: [1x1 event.listener]
```

Use the `getStatement` method to check the status:

```
getStatement(BA)
```

```
-------------------------
Account: 1234567
CurrentBalance: 500.00
Account Status: open
-------------------------
```

Make a withdrawal of $600, which results in a negative account balance:

```
withdraw(BA,600)
getStatement(BA)
```

```
-------------------------
Account: 1234567
CurrentBalance: -100.00
Account Status: overdrawn
-------------------------
```

The $600 withdrawal triggered the `InsufficientsFunds` event. The current criteria defined by the `AccountManager` class results in a status of `overdrawn`.

Make another withdrawal of $200:

```
withdraw(BA,200)
getStatement(BA)
```

```
-------------------------
Account: 1234567
CurrentBalance: -300.00
```

```
Account Status: closed
-------------------------
```

Now the `AccountStatus` has been set to `closed` by the listener and further attempts to make withdrawals are blocked without triggering the event:

```
withdraw(BA,100)
```

```
Account 1234567 has been closed.
```

If the `AccountBalance` is returned to a positive value by a deposit, then the `AccountStatus` is returned to open and withdrawals are allowed again:

```
deposit(BA,700)
getStatement(BA)
```

```
-------------------------
Account: 1234567
CurrentBalance: 400.00
Account Status: open
-------------------------
```

Representing Structured Data with Classes

In this section...

Objects as Data Structures

This example defines a class for storing data with a specific structure. Using a consistent structure for data storage makes it easier to create functions that operate on the data. A MATLAB `struct` with field names describing the particular data element is a useful way to organize data. However, a class can define both the data storage (properties) and operations that you can perform on that data (methods). This example illustrates these advantages.

Background for the Example

For this example, the data represents tensile stress/strain measurements. These data are used to calculate the elastic modulus of various materials. In simple terms, stress is the force applied to a material and strain is the resulting deformation. Their ratio defines a characteristic of the material. While this approach is an over simplification of the process, it suffices for this example.

Structure of the Data

This table describes the structure of the data.

Data	Description
`Material`	`char` vector identifying the type of material tested
`SampleNumber`	Number of a particular test sample
`Stress`	Vector of numbers representing the stress applied to the sample during the test.
`Strain`	Vector of numbers representing the strain at the corresponding values of the applied stress.
`Modulus`	Number defining an elastic modulus of the material under test, which is calculated from the stress and strain data

The TensileData Class

This example begins with a simple implementation of the class and builds on this implementation to illustrate how features enhance the usefulness of the class.

The first version of the class provides only data storage. The class defines a property for each of the required data elements.

```
classdef TensileData
   properties
      Material
      SampleNumber
      Stress
      Strain
      Modulus
   end
end
```

Create an Instance and Assign Data

The following statements create a `TensileData` object and assign data to it:

```
td = TensileData;
td.Material = 'Carbon Steel';
td.SampleNumber = 001;
td.Stress = [2e4 4e4 6e4 8e4];
td.Strain = [.12 .20 .31 .40];
td.Modulus = mean(td.Stress./td.Strain);
```

Advantages of a Class vs. a Structure

Treat the `TensileData` object (`td` in the previous statements) much as you would any MATLAB structure. However, defining a specialized data structure as a class has advantages over using a general-purpose data structure, like a MATLAB `struct`:

- Users cannot accidentally misspell a field name without getting an error. For example, typing the following:

  ```
  td.Modulus = ...
  ```

  would simply add a field to a structure. However, it returns an error when `td` is an instance of the `TensileData` class.
- A class is easy to reuse. Once you have defined the class, you can easily extend it with subclasses that add new properties.
- A class is easy to identify. A class has a name so that you can identify objects with the `whos` and `class` functions and the Workspace browser. The class name makes it easy to refer to records with a meaningful name.
- A class can validate individual field values when assigned, including class or value.
- A class can restrict access to fields, for example, allowing a particular field to be read, but not changed.

Restrict Properties to Specific Values

Restrict properties to specific values by defining a property set access method. MATLAB calls the set access method whenever setting a value for a property.

Material Property Set Function

The `Material` property set method restricts the assignment of the property to one of the following strings: `aluminum`, `stainless steel`, or `carbon steel`.

Add this function definition to the methods block.

```
classdef TensileData
   properties
      Material
      SampleNumber
      Stress
      Strain
      Modulus
   end
   methods
      function obj = set.Material(obj,material)
         if (strcmpi(material,'aluminum') ||...
               strcmpi(material,'stainless steel') ||...
               strcmpi(material,'carbon steel'))
            obj.Material = material;
         else
            error('Invalid Material')
         end
      end
   end
end
```

When there is an attempt to set the `Material` property, MATLAB calls the `set.Material` method before setting the property value.

If the value matches the acceptable values, the function set the property to that value. The code within set method can access the property directly to avoid calling the property set method recursively.

For example:

```
td = TensileData;
td.Material = 'brass';
```

```
Error using TensileData/set.Material
Invalid Material
```

Simplifying the Interface with a Constructor

Simplify the interface to the `TensileData` class by adding a constructor that:

- Enables you to pass the data as arguments to the constructor
- Assigns values to properties

The constructor is a method having the same name as the class.

```
methods
   function td = TensileData(material,samplenum,stress,strain)
      if nargin > 0
```

```
            td.Material = material;
            td.SampleNumber = samplenum;
            td.Stress = stress;
            td.Strain = strain;
        end
    end
end
```

Create a `TensileData` object fully populated with data using the following statement:

```
td = TensileData('carbon steel',1,...
      [2e4 4e4 6e4 8e4],...
      [.12 .20 .31 .40]);
```

Calculate Data on Demand

If the value of a property depends on the values of other properties, define that property using the `Dependent` attribute. MATLAB does not store the values of dependent properties. The dependent property get method determines the property value when the property is accessed. Access can occur when displaying object properties or as the result of an explicit query.

Calculating Modulus

`TensileData` objects do not store the value of the `Modulus` property. The constructor does not have an input argument for the value of the `Modulus` property. The value of the `Modulus`:

- Is calculated from the `Stress` and `Strain` property values
- Must change if the value of the `Stress` or `Strain` property changes

Therefore, it is better to calculate the value of the `Modulus` property only when its value is requested. Use a property `get` access method to calculate the value of the `Modulus`.

Modulus Property Get Method

The `Modulus` property depends on `Stress` and `Strain`, so its `Dependent` attribute is `true`. Place the `Modulus` property in a separate `properties` block and set the `Dependent` attribute.

The `get.Modulus` method calculates and returns the value of the `Modulus` property.

```
properties (Dependent)
   Modulus
end
```

Define the property get method in a `methods` block using only default attributes.

```
methods
   function modulus = get.Modulus(obj)
      ind = find(obj.Strain > 0);
      modulus = mean(obj.Stress(ind)./obj.Strain(ind));
   end
end
```

This method calculates the average ratio of stress to strain data after eliminating zeros in the denominator data.

MATLAB calls the `get.Modulus` method when the property is queried. For example,

```
td = TensileData('carbon steel',1,...
      [2e4 4e4 6e4 8e4],...
      [.12 .20 .31 .40]);
td.Modulus

ans =
  1.9005e+005
```

Modulus Property Set Method

To set the value of a `Dependent` property, the class must implement a property set method. There is no need to allow explicit setting of the `Modulus` property. However, a set method enables you to provide a customized error message. The `Modulus` set method references the current property value and then returns an error:

```
methods
   function obj = set.Modulus(obj,~)
      fprintf('%s%d\n','Modulus is: ',obj.Modulus)
      error('You cannot set the Modulus property');
   end
end
```

Displaying TensileData Objects

The `TensileData` class overloads the `disp` method. This method controls object display in the command window.

The `disp` method displays the value of the `Material`, `SampleNumber`, and `Modulus` properties. It does not display the `Stress` and `Strain` property data. These properties contain raw data that is not easily viewed in the command window.

The `disp` method uses `fprintf` to display formatted text in the command window:

```
methods
   function disp(td)
      fprintf(1,...
         'Material: %s\nSample Number: %g\nModulus: %1.5g\n',...
         td.Material,td.SampleNumber,td.Modulus);
   end
end
```

Method to Plot Stress vs. Strain

It is useful to view a graph of the stress/strain data to determine the behavior of the material over a range of applied tension. The `TensileData` class overloads the MATLAB `plot` function.

The `plot` method creates a linear graph of the stress versus strain data and adds a title and axis labels to produce a standardized graph for the tensile data records:

```
methods
   function plot(td,varargin)
      plot(td.Strain,td.Stress,varargin{:})
      title(['Stress/Strain plot for Sample',...
```

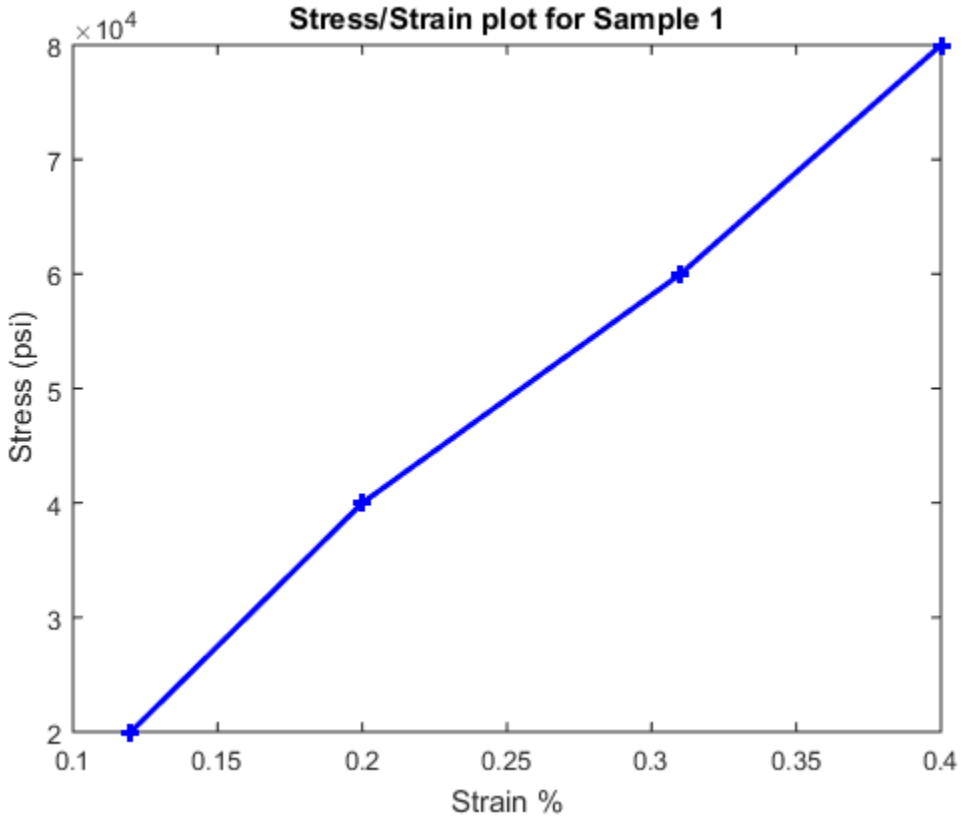
```
            num2str(td.SampleNumber)])
        ylabel('Stress (psi)')
        xlabel('Strain %')
    end
end
```

The first argument to this method is a `TensileData` object, which contains the data.

The method passes a variable list of arguments (`varargin`) directly to the built-in `plot` function. The `TensileData` `plot` method allows you to pass line specifier arguments or property name-value pairs.

For example:

```
td = TensileData('carbon steel',1,...
      [2e4 4e4 6e4 8e4],[.12 .20 .31 .40]);
plot(td,'-+b','LineWidth',2)
```



TensileData Class Synopsis

Example Code	Discussion
`classdef TensileData`	Value class enables independent copies of object. For more information, see “Comparison of Handle and Value Classes” on page 7-2

Example Code	Discussion

`properties`
`  Material`
`  SampleNumber`
`  Stress`
`  Strain`
`end`
	See “Structure of the Data” on page 3-14

`properties (Dependent)`
`  Modulus`
`end`
	Calculate `Modulus` when queried. For information about this code, see “Calculate Data on Demand” on page 3-17.
	
	For general information, see “Get and Set Methods for Dependent Properties” on page 8-42

`methods`	For general information about methods, see “Ordinary Methods” on page 9-6

`function td = TensileData(material,samplenum,...`
`  stress,strain)`
`  if nargin > 0`
`  td.Material = material;`
`  td.SampleNumber = samplenum;`
`  td.Stress = stress;`
`  td.Strain = strain;`
`  end`
`end`
	For information about this code, see “Simplifying the Interface with a Constructor” on page 3-16.
	
	For general information about constructors, see “Class Constructor Methods” on page 9-15

`function obj = set.Material(obj,material)`
`  if (strcmpi(material,'aluminum') ||...`
`  strcmpi(material,'stainless steel') ||...`
`  strcmpi(material,'carbon steel'))`
`  obj.Material = material;`
`  else`
`  error('Invalid Material')`
`  end`
`end`
	Restrict possible values for `Material` property.
	
	For information about this code, see “Restrict Properties to Specific Values” on page 3-15.
	
	For general information about property set methods, see “Property Get and Set Methods” on page 8-38.

`function m = get.Modulus(obj)`
`  ind = find(obj.Strain > 0);`
`  m = mean(obj.Stress(ind)./obj.Strain(ind));`
`end`
	Calculate `Modulus` property when queried.
	
	For information about this code, see “Modulus Property Get Method” on page 3-17.
	
	For general information about property get methods, see “Property Get and Set Methods” on page 8-38.

`function obj = set.Modulus(obj,~)`
`  fprintf('%s%d\n','Modulus is: ',obj.Modulus)`
`  error('You cannot set Modulus property');`
`end`
	Add set method for `Dependent Modulus` property. For information about this code, see “Modulus Property Set Method” on page 3-18.
	
	For general information about property set methods, see “Property Get and Set Methods” on page 8-38.

Example Code	Discussion

`function disp(td)`
`  fprintf(1,'Material: %s\nSample Number: %g\nModulus: %1.5g\n',...`
`  td.Material,td.SampleNumber,td.Modulus)`
`end`
	Overload `disp` method to display certain properties.
	
	For information about this code, see "Displaying TensileData Objects" on page 3-18
	
	For general information about overloading disp, see "Overloading the disp Function" on page 18-34

`function plot(td,varargin)`
`  plot(td.Strain,td.Stress,varargin{:})`
`  title(['Stress/Strain plot for Sample',...`
`  num2str(td.SampleNumber)])`
`  ylabel('Stress (psi)')`
`  xlabel('Strain %')`
`end`
	Overload `plot` function to accept `TensileData` objects and graph stress vs. strain.
	
	"Method to Plot Stress vs. Strain" on page 3-18

`end` `end`	`end` statements for `methods` and for `classdef`.

Expand for Class Code

```
classdef TensileData
   properties
      Material
      SampleNumber
      Stress
      Strain
   end
   properties (Dependent)
      Modulus
   end

   methods
      function td = TensileData(material,samplenum,stress,strain)
         if nargin > 0
            td.Material = material;
            td.SampleNumber = samplenum;
            td.Stress = stress;
            td.Strain = strain;
         end
      end

      function obj = set.Material(obj,material)
         if (strcmpi(material,'aluminum') ||...
               strcmpi(material,'stainless steel') ||...
               strcmpi(material,'carbon steel'))
            obj.Material = material;
         else
            error('Invalid Material')
         end
      end

      function m = get.Modulus(obj)
         ind = find(obj.Strain > 0);
         m = mean(obj.Stress(ind)./obj.Strain(ind));
      end
```

```
      function obj = set.Modulus(obj,~)
         fprintf('%s%d\n','Modulus is: ',obj.Modulus)
         error('You cannot set Modulus property');
      end

      function disp(td)
         sprintf('Material: %s\nSample Number: %g\nModulus: %1.5g\n',...
            td.Material,td.SampleNumber,td.Modulus)
      end

      function plot(td,varargin)
         plot(td.Strain,td.Stress,varargin{:})
         title(['Stress/Strain plot for Sample ',...
            num2str(td.SampleNumber)])
         xlabel('Strain %')
         ylabel('Stress (psi)')
      end
   end
end
```

See Also

More About

- “Components of a Class” on page 5-2

Implementing Linked Lists with Classes

In this section...

Class Definition Code

For the class definition code listing, see “dlnode Class Synopsis” on page 3-25.

To use the class, create a folder named `@dlnode` and save `dlnode.m` to this folder. The parent folder of `@dlnode` must be on the MATLAB path. Alternatively, save `dlnode.m` to a path folder.

dlnode Class Design

`dlnode` is a class for creating doubly linked lists in which each node contains:

- Data array
- Handle to the next node
- Handle to the previous node

Each node has methods that enable the node to be:

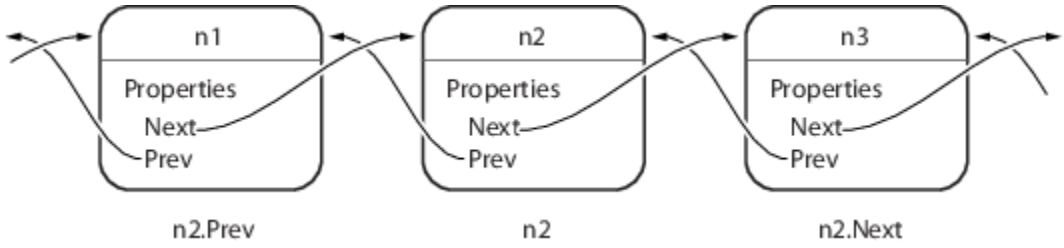
- Inserted before a specified node in a linked list
- Inserted after a specific node in a linked list
- Removed from a list

Class Properties

The `dlnode` class implements each node as a handle object with three properties:

- `Data` — Contains the data for this node
- `Next` — Contains the handle of the next node in the list (`SetAccess = private`)
- `Prev` — Contains the handle of the previous node in the list (`SetAccess = private`)

This diagram shows a list with three-nodes `n1`, `n2`, and `n3`. It also shows how the nodes reference the next and previous nodes.



Class Methods

The `dlnode` class implements the following methods:

- `dlnode` — Construct a node and assign the value passed as an input to the `Data` property
- `insertAfter` — Insert this node after the specified node
- `insertBefore` — Insert this node before the specified node
- `removeNode` — Remove this node from the list and reconnect the remaining nodes
- `clearList` — Remove large lists efficiently
- `delete` — Private method called by MATLAB when deleting the list.

Create Doubly Linked List

Create a node by passing the node's data to the `dlnode` class constructor. For example, these statements create three nodes with data values `1`, `2`, and `3`:

```
n1 = dlnode(1);
n2 = dlnode(2);
n3 = dlnode(3);
```

Build these nodes into a doubly linked list using the class methods designed for this purpose:

```
n2.insertAfter(n1) % Insert n2 after n1
n3.insertAfter(n2) % Insert n3 after n2
```

Now the three nodes are linked:

```
n1.Next % Points to n2

ans =

  dlnode with properties:

    Data: 2
    Next: [1x1 dlnode]
    Prev: [1x1 dlnode]

n2.Next.Prev % Points back to n2

ans =

  dlnode with properties:

    Data: 2
    Next: [1x1 dlnode]
    Prev: [1x1 dlnode]

n1.Next.Next % Points to n3

ans =

  dlnode with properties:

    Data: 3
    Next: []
    Prev: [1x1 dlnode]
```

```
n3.Prev.Prev % Points to n1

ans =

  dlnode with properties:

    Data: 1
    Next: [1x1 dlnode]
    Prev: []
```

Why a Handle Class for Linked Lists?

Each node is unique in that no two nodes can be previous to or next to the same node.

For example, a node object, `node`, contains in its `Next` property the handle of the next node object, `node.Next`. Similarly, the `Prev` property contains the handle of the previous node, `node.Prev`. Using the three-node linked list defined in the previous section, you can demonstrate that the following statements are true:

```
n1.Next == n2
n2.Prev == n1
```

Now suppose that you assign `n2` to `x`:

```
x = n2;
```

The following two equalities are then true:

```
x == n1.Next
x.Prev == n1
```

But each instance of a node is unique so there is only one node in the list that can satisfy the conditions of being equal to `n1.Next` and having a `Prev` property that contains a handle to `n1`. Therefore, `x` must point to the same node as `n2`.

There has to be a way for multiple variables to refer to the same object. The MATLAB `handle` class provides a means for both `x` and `n2` to refer to the same node.

The handle class defines the `eq` method (use `methods('handle')` to list the handle class methods), which enables the use of the `==` operator with all handle objects.

Related Information

For more information on handle classes, see “Comparison of Handle and Value Classes” on page 7-2.

dlnode Class Synopsis

This section describes the implementation of the `dlnode` class.

Example Code	Discussion
`classdef dlnode < handle`	"dlnode Class Design" on page 3-23 "Why a Handle Class for Linked Lists?" on page 3-25 "Comparison of Handle and Value Classes" on page 7-2

`properties`
`  Data`
`end`
	"dlnode Class Design" on page 3-23

`properties (SetAccess = private)`
`  Next = dlnode.empty`
`  Prev = dlnode.empty`
`end`
	"Property Attributes" on page 8-8: `SetAccess`.
	
	Initialize these properties to `empty dlnode` objects.
	
	For general information about properties, see "Property Syntax" on page 8-4

`methods`	For general information about methods, see "Methods in Class Design" on page 9-2

`function node = dlnode(Data)`
`  if (nargin > 0)`
`  node.Data = Data;`
`  end`
`end`
	Creating an individual node (not connected) requires only the data.
	
	For general information about constructors, see "Guidelines for Constructors" on page 9-16

`function insertAfter(newNode, nodeBefore)`
`  removeNode(newNode);`
`  newNode.Next = nodeBefore.Next;`
`  newNode.Prev = nodeBefore;`
`  if ~isempty(nodeBefore.Next)`
`  nodeBefore.Next.Prev = newNode;`
`  end`
`  nodeBefore.Next = newNode;`
`end`
	Insert node into a doubly linked list after specified node, or link the two specified nodes if there is not already a list. Assigns the correct values for `Next` and `Prev` properties.
	
	"Insert Nodes" on page 3-29

`function insertBefore(newNode, nodeAfter)`
`  removeNode(newNode);`
`  newNode.Next = nodeAfter;`
`  newNode.Prev = nodeAfter.Prev;`
`  if ~isempty(nodeAfter.Prev)`
`  nodeAfter.Prev.Next = newNode;`
`  end`
`  nodeAfter.Prev = newNode;`
`end`
	Insert node into doubly linked list before specified node, or link the two specified nodes if there is not already a list. This method assigns correct values for `Next` and `Prev` properties.
	
	See "Insert Nodes" on page 3-29

`function removeNode(node)`
`  if ~isscalar(node)`
`  error('Nodes must be scalar')`
`  end`
`  prevNode = node.Prev;`
`  nextNode = node.Next;`
`  if ~isempty(prevNode)`
`  prevNode.Next = nextNode;`
`  end`
`  if ~isempty(nextNode)`
`  nextNode.Prev = prevNode;`
`  end`
`  node.Next =  = dlnode.empty;`
`  node.Prev =  = dlnode.empty;`
`end`
	Remove node and fix the list so that remaining nodes are properly connected. `node` argument must be scalar.
	
	Once there are no references to node, MATLAB deletes it.
	
	"Remove a Node" on page 3-30

Example Code	Discussion

`function clearList(node)`
`  prev = node.Prev;`
`  next = node.Next;`
`  removeNode(node)`
`  while ~isempty(next)`
`  node = next;`
`  next = node.Next;`
`  removeNode(node);`
`  end`
`  while ~isempty(prev)`
`  node = prev;`
`  prev = node.Prev;`
`  removeNode(node)`
`  end`
`end`
	Avoid recursive calls to destructor as a result of clearing the list variable. Loop through list to disconnect each node. When there are no references to a node, MATLAB calls the class destructor (see the `delete` method) before deleting it.

`methods (Access = private)`
`  function delete(node)`
`  clearList(node)`
`end`
	Class destructor method. MATLAB calls the `delete` method you delete a node that is still connected to the list.

`end` `end`	End of private methods and end of class definition.

Expand for Class Code

```
classdef dlnode < handle
   % dlnode A class to represent a doubly-linked node.
   % Link multiple dlnode objects together to create linked lists.
   properties
      Data
   end
   properties(SetAccess = private)
      Next = dlnode.empty
      Prev = dlnode.empty
   end

   methods
      function node = dlnode(Data)
         % Construct a dlnode object
         if nargin > 0
            node.Data = Data;
         end
      end

      function insertAfter(newNode, nodeBefore)
         % Insert newNode after nodeBefore.
         removeNode(newNode);
         newNode.Next = nodeBefore.Next;
         newNode.Prev = nodeBefore;
         if ~isempty(nodeBefore.Next)
            nodeBefore.Next.Prev = newNode;
         end
         nodeBefore.Next = newNode;
      end

      function insertBefore(newNode, nodeAfter)
         % Insert newNode before nodeAfter.
         removeNode(newNode);
         newNode.Next = nodeAfter;
         newNode.Prev = nodeAfter.Prev;
```

```
            if ~isempty(nodeAfter.Prev)
               nodeAfter.Prev.Next = newNode;
            end
            nodeAfter.Prev = newNode;
         end

         function removeNode(node)
            % Remove a node from a linked list.
            if ~isscalar(node)
               error('Input must be scalar')
            end
            prevNode = node.Prev;
            nextNode = node.Next;
            if ~isempty(prevNode)
               prevNode.Next = nextNode;
            end
            if ~isempty(nextNode)
               nextNode.Prev = prevNode;
            end
            node.Next = dlnode.empty;
            node.Prev = dlnode.empty;
         end

         function clearList(node)
            % Clear the list before
            % clearing list variable
            prev = node.Prev;
            next = node.Next;
            removeNode(node)
            while ~isempty(next)
               node = next;
               next = node.Next;
               removeNode(node);
            end
            while ~isempty(prev)
               node = prev;
               prev = node.Prev;
               removeNode(node)
            end
         end
      end

      methods (Access = private)
         function delete(node)
            clearList(node)
         end
      end
   end
```

Class Properties

Only `dlnode` class methods can set the `Next` and `Prev` properties because these properties have private set access (`SetAccess = private`). Using private set access prevents client code from performing any incorrect operation with these properties. The `dlnode` class methods perform all the operations that are allowed on these nodes.

The `Data` property has public set and get access, allowing you to query and modify the value of `Data` as required.

Here is how the `dlnode` class defines the properties:

```
properties
   Data
end
properties(SetAccess = private)
   Next = dlnode.empty;
   Prev = dlnode.empty;
end
```

Construct a Node Object

To create a node object, specify the node's data as an argument to the constructor:

```
function node = dlnode(Data)
   if nargin > 0
      node.Data = Data;
   end
end
```

Insert Nodes

There are two methods for inserting nodes into the list — `insertAfter` and `insertBefore`. These methods perform similar operations, so this section describes only `insertAfter` in detail.

```
function insertAfter(newNode, nodeBefore)
   removeNode(newNode);
   newNode.Next = nodeBefore.Next;
   newNode.Prev = nodeBefore;
   if ~isempty(nodeBefore.Next)
      nodeBefore.Next.Prev = newNode;
   end
   nodeBefore.Next = newNode;
end
```

How insertAfter Works

First, `insertAfter` calls the `removeNode` method to ensure that the new node is not connected to any other nodes. Then, `insertAfter` assigns the `newNode` `Next` and `Prev` properties to the handles of the nodes that are after and before the `newNode` location in the list.

For example, suppose that you want to insert a new node, `nnew`, after an existing node, `n1`, in a list containing `n1`—`n2`—`n3`.

First, create `nnew`:

```
nnew = dlnode(rand(3));
```

Next, call `insertAfter` to insert `nnew` into the list after `n1`:

```
nnew.insertAfter(n1)
```

The `insertAfter` method performs the following steps to insert `nnew` in the list between `n1` and `n2`:

- Set `nnew.Next` to `n1.Next` (`n1.Next` is `n2`):

  ```
  nnew.Next = n1.Next;
  ```

- Set `nnew.Prev` to `n1`

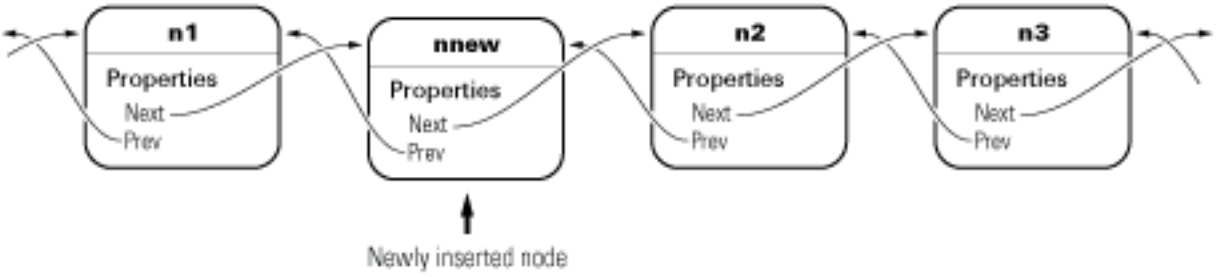
  ```
  nnew.Prev = n1;
  ```

- If `n1.Next` is not empty, then `n1.Next` is still `n2`, so `n1.Next.Prev` is `n2.Prev`, which is set to `nnew`

  ```
  n1.Next.Prev = nnew;
  ```

- `n1.Next` is now set to `nnew`

  ```
  n1.Next = nnew;
  ```



Remove a Node

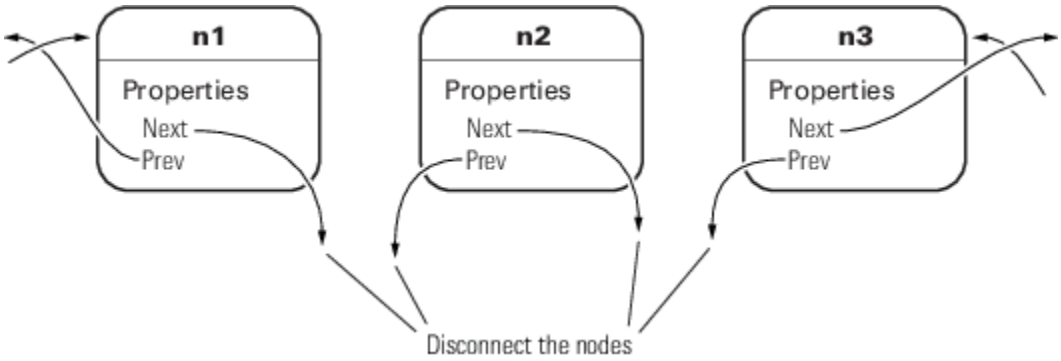
The `removeNode` method removes a node from a list and reconnects the remaining nodes. The `insertBefore` and `insertAfter` methods always call `removeNode` on the node to insert before attempting to connect it to a linked list.

Calling `removeNode` ensures that the node is in a known state before assigning it to the `Next` or `Prev` property:

```
function removeNode(node)
   if ~isscalar(node)
      error('Input must be scalar')
   end
   prevNode = node.Prev;
   nextNode = node.Next;
   if ~isempty(prevNode)
      prevNode.Next = nextNode;
   end
   if ~isempty(nextNode)
      nextNode.Prev = prevNode;
   end
   node.Next = dlnode.empty;
   node.Prev = dlnode.empty;
end
```

For example, suppose that you remove `n2` from a three-node list (`n1`–`n2`–`n3`):

```
n2.removeNode;
```



`removeNode` removes `n2` from the list and reconnects the remaining nodes with the following steps:

```
n1 = n2.Prev;
n3 = n2.Next;
if n1 exists, then
   n1.Next = n3;
if n3 exists, then
   n3.Prev = n1
```

The list is rejoined because `n1` connects to `n3` and `n3` connects to `n1`. The final step is to ensure that `n2.Next` and `n2.Prev` are both empty (that is, `n2` is not connected):

```
n2.Next = dlnode.empty;
n2.Prev = dlnode.empty;
```

Removing a Node from a List

Suppose that you create a list with 10 nodes and save the handle to the head of the list:

```
head = dlnode(1);
for i = 10:-1:2
   new = dlnode(i);
   insertAfter(new,head);
end
```

Now remove the third node (`Data` property assigned the value `3`):

```
removeNode(head.Next.Next)
```

Now the third node in the list has a data value of `4`:

```
head.Next.Next
```

```
ans =

  dlnode with properties:

    Data: 4
    Next: [1x1 dlnode]
    Prev: [1x1 dlnode]
```

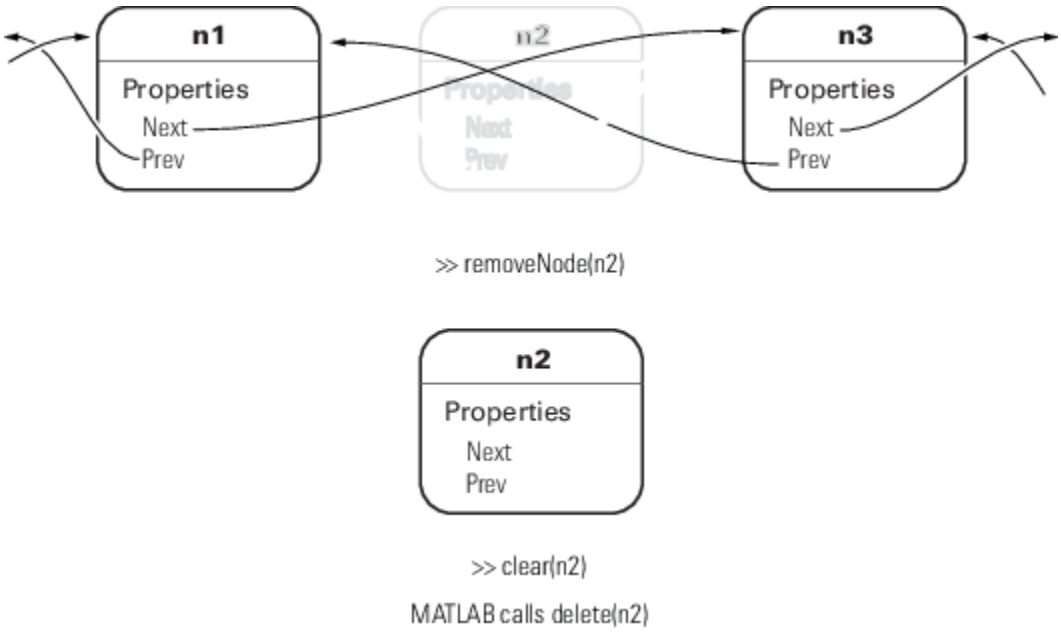
And the previous node has a `Data` value of `2`:

```
head.Next
```

```
ans =

  dlnode with properties:

    Data: 2
    Next: [1x1 dlnode]
    Prev: [1x1 dlnode]
```

Delete a Node

To delete a node, call the `removeNode` method on that node. The `removeNode` method disconnects the node and reconnects the list before allowing MATLAB to destroy the removed node. MATLAB destroys the node once references to it by other nodes are removed and the list is reconnected.



Delete the List

When you create a linked list and assign a variable that contains, for example, the head or tail of the list, clearing that variable causes the destructor to recurse through the entire list. With large enough list, clearing the list variable can result in MATLAB exceeding its recursion limit.

The `clearList` method avoids recursion and improves the performance of deleting large lists by looping over the list and disconnecting each node. `clearList` accepts the handle of any node in the list and removes the remaining nodes.

```
function clearList(node)
   if ~isscalar(node)
      error('Input must be scalar')
   end
   prev = node.Prev;
   next = node.Next;
   removeNode(node)
   while ~isempty(next)
      node = next;
      next = node.Next;
```

```
      removeNode(node);
   end
   while ~isempty(prev)
      node = prev;
      prev = node.Prev;
      removeNode(node)
   end
end
```

For example, suppose that you create a list with many nodes:

```
head = dlnode(1);
for k = 100000:-1:2
   nextNode = dlnode(k);
   insertAfter(nextNode,head)
end
```

The variable `head` contains the handle to the node at the head of the list:

```
head
```

```
head =

  dlnode with properties:

    Data: 1
    Next: [1x1 dlnode]
    Prev: []
```

```
head.Next
```

```
ans =

  dlnode with properties:

    Data: 2
    Next: [1x1 dlnode]
    Prev: [1x1 dlnode]
```

You can call `clearList` to remove the whole list:

```
clearList(head)
```

The only nodes that have not been deleted by MATLAB are those nodes for which there exists an explicit reference. In this case, those references are `head` and `nextNode`:

```
head
```

```
head =

  dlnode with properties:

    Data: 1
    Next: []
    Prev: []
```

```
nextNode
```

```
nextNode =
```

```
  dlnode with properties:

    Data: 2
    Next: []
    Prev: []
```

You can remove these nodes by clearing the variables:

```
clear head nextNode
```

The delete Method

The `delete` method simply calls the `clearList` method:

```
methods (Access = private)
   function delete(node)
      clearList(node)
   end
end
```

The `delete` method has private access to prevent users from calling `delete` when intending to delete a single node. MATLAB calls `delete` implicitly when the list is destroyed.

To delete a single node from the list, use the `removeNode` method.

Specialize the dlnode Class

The `dlnode` class implements a doubly linked list and provides a convenient starting point for creating more specialized types of linked lists. For example, suppose that you want to create a list in which each node has a name.

Rather than copying the code used to implement the `dlnode` class, and then expanding upon it, you can derive a new class from `dlnode` (that is, subclass `dlnode`). You can create a class that has all the features of `dlnode` and also defines its own additional features. And because `dlnode` is a handle class, this new class is a handle class too.

NamedNode Class Definition

To use the class, create a folder named `@NamedNode` and save `NamedNode.m` to this folder. The parent folder of `@NamedNode` must be on the MATLAB path. Alternatively, save `NamedNode.m` to a path folder.

The following class definition shows how to derive the `NamedNode` class from the `dlnode` class:

```
classdef NamedNode < dlnode
   properties
      Name = ''
   end
   methods
      function n = NamedNode (name,data)
         if nargin == 0
            name = '';
            data = [];
         end
         n = n@dlnode(data);
```

```
            n.Name = name;
        end
    end
end
```

The `NamedNode` class adds a `Name` property to store the node name.

The constructor calls the class constructor for the `dlnode` class, and then assigns a value to the `Name` property.

Use NamedNode to Create a Doubly Linked List

Use the `NamedNode` class like the `dlnode` class, except that you specify a name for each node object. For example:

```
n(1) = NamedNode('First Node',100);
n(2) = NamedNode('Second Node',200);
n(3) = NamedNode('Third Node',300);
```

Now use the insert methods inherited from `dlnode` to build the list:

```
n(2).insertAfter(n(1))
n(3).insertAfter(n(2))
```

A single node displays its name and data when you query its properties:

```
n(1).Next

ans =

  NamedNode with properties:

    Name: 'Second Node'
    Data: 200
    Next: [1x1 NamedNode]
    Prev: [1x1 NamedNode]

n(1).Next.Next

ans =

  NamedNode with properties:

    Name: 'Third Node'
    Data: 300
    Next: []
    Prev: [1x1 NamedNode]

n(3).Prev.Prev

ans =

  NamedNode with properties:

    Name: 'First Node'
    Data: 100
    Next: [1x1 NamedNode]
    Prev: []
```

See Also

More About

- “The Handle Superclass” on page 7-11

Working with Objects in MATLAB

Some MATLAB® functions return *objects*. Objects combine data (*properties*) with functions and methods. Object properties contain data, including simple types like numbers or text, or other objects. The functions and methods perform actions on the objects themselves. These functions can act on the object properties or change the state of the object, for example.

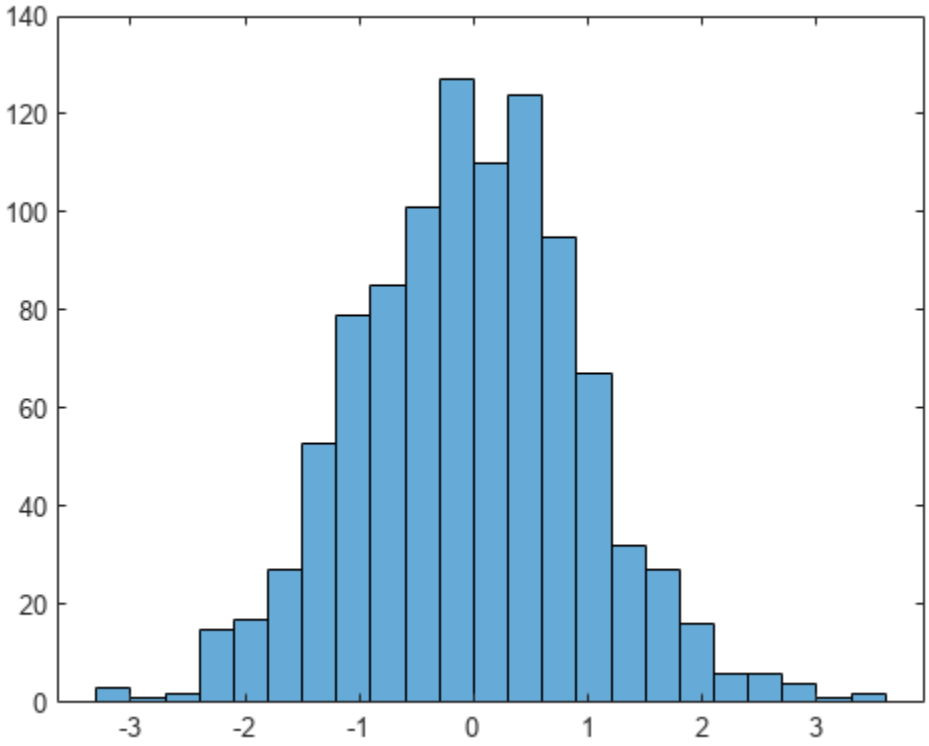
Create an Object

When creating an object, you can assign a variable to that object. The variable provides access to the properties and methods of the object. For example, this syntax for the `histogram` function not only displays a histogram of the data in `x` but also returns the object as the output `h`.

```
h = histogram(x)
```

Create a histogram object that displays 1000 random numbers. Calling `histogram` with an output argument displays the graph, the type or class of the object (`Histogram`), and a partial list of the object properties and their values.

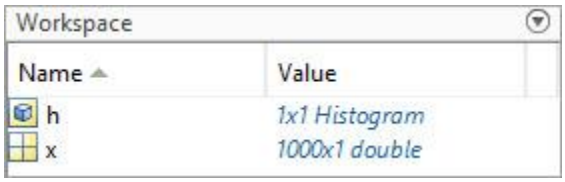
```
x = randn(1000,1);
h = histogram(x)
```




```
h =
  Histogram with properties:
```

```
            Data: [1000x1 double]
          Values: [3 1 2 15 17 27 53 79 85 101 127 110 124 95 67 32 27 16 6 6 4 1 2]
         NumBins: 23
        BinEdges: [-3.3000 -3.0000 -2.7000 -2.4000 -2.1000 -1.8000 -1.5000 -1.2000 -0.9000 -0.60
        BinWidth: 0.3000
       BinLimits: [-3.3000 3.6000]
  Normalization: 'count'
      FaceColor: 'auto'
      EdgeColor: [0 0 0]

  Show all properties
```

In the workspace, the histogram object is listed with the other active variables, including the dimensions and the type of object.



Get and Set Object Properties

Object properties contain data. By changing property values, you can modify certain aspects of an object. You can use dot notation with the variable assigned to the object to access and change its properties.

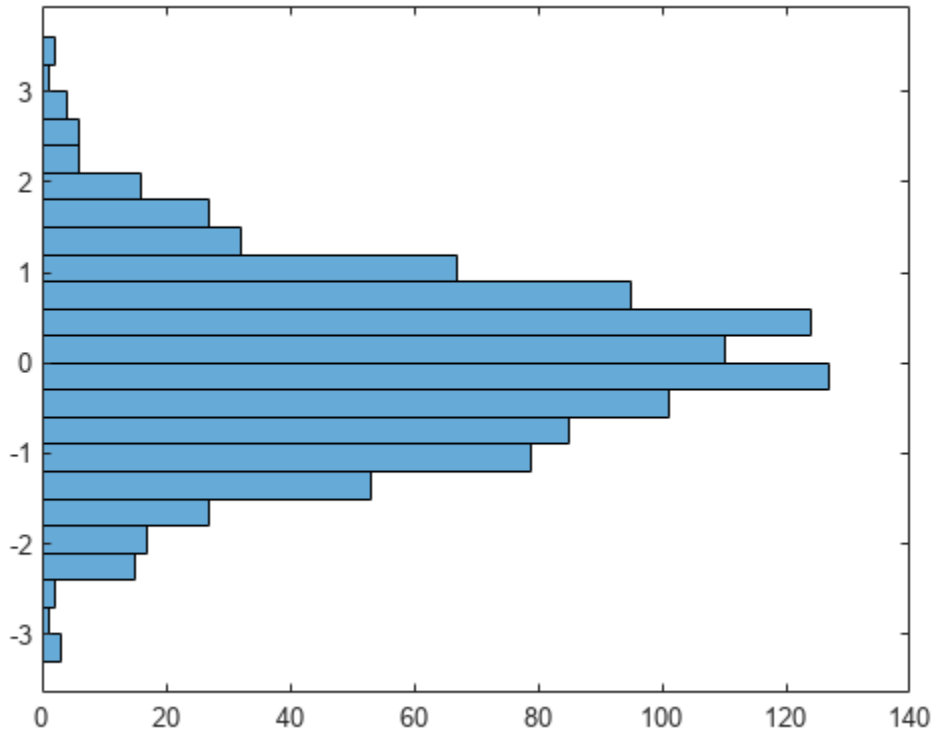
In the case of a histogram object, properties contain the raw data, the number of bins, the height of each bar, and other information that controls the appearance of the histogram. For example, `Orientation` is a property of histogram objects that determines whether the bars are displayed horizontally or vertically. You can access the value of the property by entering the name of the object (`h`), a dot, and the property name.

```
h.Orientation
```

```
ans =
'vertical'
```

You can change the value of the property using the same syntax, but with the addition of an equal sign and the new value.

```
h.Orientation = "horizontal"
```




```
h =
  Histogram with properties:

            Data: [1000x1 double]
          Values: [3 1 2 15 17 27 53 79 85 101 127 110 124 95 67 32 27 16 6 6 4 1 2]
         NumBins: 23
        BinEdges: [-3.3000 -3.0000 -2.7000 -2.4000 -2.1000 -1.8000 -1.5000 -1.2000 -0.9000 -0.60
        BinWidth: 0.3000
       BinLimits: [-3.3000 3.6000]
    Normalization: 'count'
        FaceColor: 'auto'
        EdgeColor: [0 0 0]

  Show all properties
```

Not all object properties are writeable. A property can be read-only, meaning that it can be read, but trying to assign a new value to it returns an error. For example, the `Values` property of a histogram stores the height of each bar and is calculated when the object is created. It is a read-only property, so you cannot change `Values` directly.

Functions That Accept Objects

Some functions are designed to perform actions on an object. These functions can be *methods*, which are defined specifically for one class of objects, or can be functions that accept that object as an ordinary input argument. In either case, you can use the variable for the object as an input. For example, `Histogram` objects include the functions `morebins` and `fewerbins`, which increase and

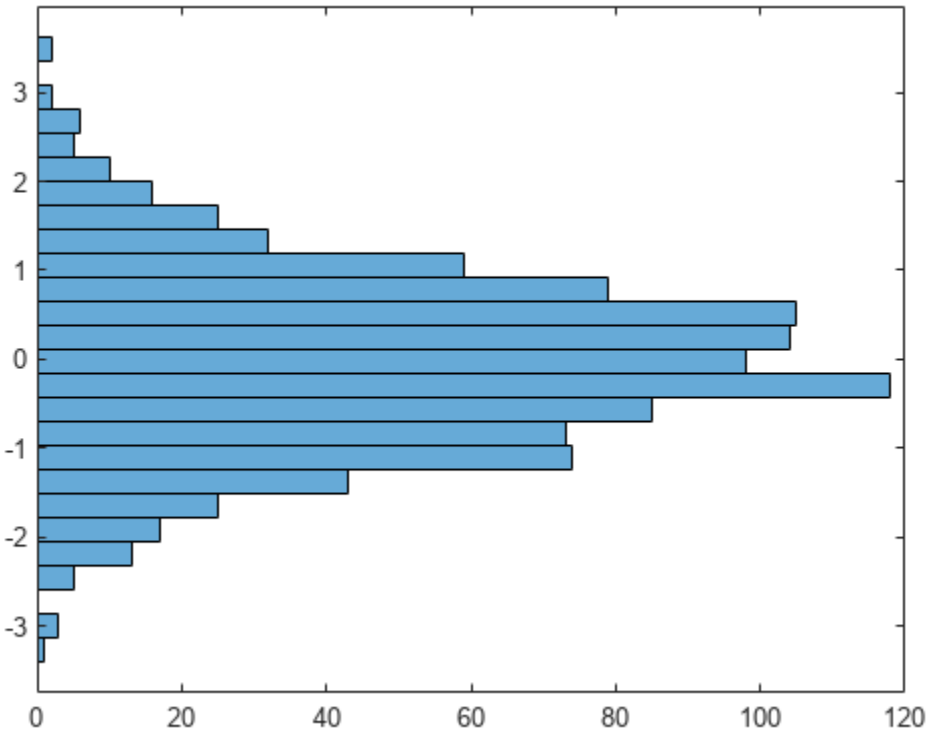
decrease the number of histogram bins, respectively. Access the `NumBins` property to see how many bins are currently in the histogram.

```
h.NumBins
```

```
ans = 23
```

Call `morebins` to increase the number of bins in histogram `h`. The `morebins` function increases the number of bins by approximately 10%, so the number of bins increases from 23 to 26.

```
morebins(h)
```




```
ans = 26
```

Other object-oriented programming languages frequently use dot notation to call methods, such as `h.morebins`. That syntax also is supported for methods in MATLAB. However, for consistency in sample code, the documentation uses function form for most methods and functions that accept object inputs.

The `morebins` function does not have any additional input arguments, but that is not true of all functions that accept objects. Besides the object itself, object functions can have additional input arguments, which you can pass to the function with standard function syntax. For example, you can call to an object function that takes two input arguments with this syntax:

```
functionName(objectVariable,arg1,arg2)
```

Define Your Own Class-Based Objects

In addition to the objects provided in the MATLAB language, you can define your own class-based objects using object-oriented programming techniques. The language follows standard OO conventions. For more information, start with “Creating a Simple Class” on page 2-2.

See Also

“Fundamental MATLAB Classes” | “Creating a Simple Class” on page 2-2

Static Data

Static Data

In this section...

What Is Static Data

Static data refers to data that all objects of the class share and that you can modify after creation. Use static data to define counters used by class instances or other data that is shared among all objects of a class. Unlike instance data, static data does not vary from one object to another. MATLAB provides several ways to define static data, depending on your requirements.

Static Variable

Classes can use a persistent variable to store static data. Define a static method or local function in which you create a persistent variable. The method or function provides access to this variable. Use this technique when you want to store one or two variables.

Saving an object of the class defining the persistent variable does not save the static data associated with the class. To save your static data in an object, or define more extensive data, use the static data object technique “Static Data Object” on page 4-3

Implementation

The `StoreData` class defines a static method that declares a persistent variable `Var`. The `setgetVar` method provides set and get access to the data in the persistent variable. Because the `setgetVar` method has public access, you can set and get the data stored in the persistent variable globally. Control the scope of access by setting the method `Access` attribute.

```
classdef StoreData
   methods (Static)
      function out = setgetVar(data)
         persistent Var;
         if nargin
            Var = data;
         end
         out = Var;
      end
   end
end
```

Set the value of the variable by calling `setgetVar` with an input argument. The method assigns the input value to the persistent variable:

```
StoreData.setgetVar(10);
```

Get the value of the variable by calling `setgetVar` with no input argument:

```
a = StoreData.setgetVar
```

```
a =

    10
```

Clear the persistent variable by calling `clear` on the `StoreData` class:

```
clear StoreData
a = StoreData.setgetVar
```

```
a =

     []
```

Add a method like `setgetVar` to any class in which you want the behavior of a static property.

Static Data Object

To store more extensive data, define a handle class with public properties. Assign an object of the class to a constant property of the class that uses the static data. This technique is useful when you want to:

- Add more properties or methods that modify the data.
- Save objects of the data class and reload the static data.

Implementation

The `SharedData` class is a handle class, which enables you to reference the same object data from multiple handle variables:

```
classdef SharedData < handle
   properties
      Data1
      Data2
   end
end
```

The `UseData` class is the stub of a class that uses the data stored in the `SharedData` class. The `UseData` class stores the handle to a `SharedData` object in a constant property.

```
classdef UseData
   properties (Constant)
      Data = SharedData
   end
   % Class code here
end
```

The `Data` property contains the handle of the `SharedData` object. MATLAB constructs the `SharedData` object when loading the `UseData` class. All subsequently created instances of the `UseData` class refer to the same `SharedData` object.

To initialize the `SharedData` object properties, load the `UseData` class by referencing the constant property.

```
h = UseData.Data
```

```
h =
```

```
  SharedData with properties:

    Data1: []
    Data2: []
```

Use the handle to the `SharedData` object to assign data to property values:

```
h.Data1 = 'MyData1';
h.Data2 = 'MyData2';
```

Each instance of the `UseData` class refers to the same handle object:

```
a1 = UseData;
a2 = UseData;
```

Reference the data using the object variable:

```
a1.Data.Data1
```

```
ans =

MyData1
```

Assign a new value to the properties in the `SharedData` object:

```
a1.Data.Data1 = rand(3);
```

All new and existing objects of the `UseData` class share the same `SharedData` object. `a2` now has the `rand(3)` data that was assigned to `a1` in the previous step:

```
a2.Data.Data1
```

```
ans =

    0.8147    0.9134    0.2785
    0.9058    0.6324    0.5469
    0.1270    0.0975    0.9575
```

To reinitialize the constant property, clear all instances of the `UseData` class and then clear the class:

```
clear a1 a2
clear UseData
```

Constant Data

To store constant values that do not change, assign the data to a constant property. All instances of the class share the same value for that property. Control the scope of access to constant properties by setting the property `Access` attribute.

The only way to change the value of a constant property is to change the class definition. Use constant properties like public final static fields in Java®.

See Also

`persistent` | `clear`

Related Examples

- “Define Class Properties with Constant Values” on page 15-2
- “Static Methods” on page 9-23

More About

- “Method Attributes” on page 9-4
- “Property Attributes” on page 8-8
- “Static Properties” on page 5-38

Class Definition—Syntax Reference

Components of a Class

In this section...

Class Building Blocks

MATLAB organizes class definition code into modular blocks, delimited by keywords. All keywords have an associated `end` statement:

- `classdef...end` — Definition of all class components
- `properties...end` — Declaration of property names, specification of property attributes, assignment of default values
- `methods...end` — Declaration of method signatures, method attributes, and function code
- `events...end` — Declaration of event name and attributes
- `enumeration...end` — Declaration of enumeration members and enumeration values for enumeration classes

`properties`, `methods`, `events`, and `enumeration` are keywords only within a `classdef` block.

Class Definition Block

The `classdef` block contains the class definition within a file that starts with the `classdef` keyword and terminates with the `end` keyword.

```
classdef (ClassAttributes) ClassName < SuperClass
   ...
end
```

For example, this `classdef` defines a class called `MyClass` that subclasses the `handle` class. The class is also defined as sealed, so you cannot use inherit from this class.

```
classdef (Sealed) MyClass < handle
   ...
end
```

See `classdef` for more syntax information.

Properties Block

A `properties` block contains property definitions, including optional initial values. Use a separate block for each unique set of attribute specifications. Each properties block starts with the `properties` keyword and terminates with the `end` keyword.

```
properties (PropertyAttributes)
   PropertyName size class {validators} = DefaultValue
end
```

For example, this class defines a private property `Prop1` of type `double` with a default value.

```
classdef MyClass
   properties (SetAccess = private)
      Prop1 double = 12
   end
   ...
end
```

See “Initialize Property Values” on page 8-13 for more information.

Methods Block

A `methods` block contains function definitions for the class methods. Use a separate block for each unique set of attribute specifications. Each methods block starts with the `methods` keyword and terminates with the `end` keyword.

```
methods (MethodAttributes)
   function obj = MethodName(arg1,...)
         ...
   end
```

For example, this class defines a protected method `MyMethod`.

```
classdef MyClass
   methods (Access = protected)
      function obj = myMethod(obj,arg1)
         ...
      end
   end
end
```

See “Method Syntax” on page 5-7 for more information.

MATLAB differs from languages like C++ and Java in that you must explicitly pass an object of the class to the method.

Using the `MyClass` example, call `MyMethod` using the object `obj` of the class and either function or dot syntax:

```
obj = MyClass;
r = MyMethod(obj,arg1);
r = obj.MyMethod(arg1);
```

For more information, see “Method Invocation” on page 9-11.

Events Block

The `events` block (one for each unique set of attribute specifications) contains the names of events that this class declares. The `events` block starts with the `events` keyword and terminates with the `end` keyword.

```
classdef ClassName
   events (EventAttributes)
      EventName
   end
   ...
end
```

For example, this class defined an event called `StateChange` with `ListenAccess` set to `protected`.

```
classdef EventSource
   events (ListenAccess = protected)
      StateChanged
   end
   ...
end
```

See “Events” for more information.

Attribute Specification

Attribute Syntax

Attributes modify the behavior of classes and class components (properties, methods, and events). Attributes enable you to define useful behaviors without writing complicated code. For example, you can create a read-only property by setting its `SetAccess` attribute to private but leaving its `GetAccess` attribute set to public.

```
properties (SetAccess = private)
   ScreenSize = getScreenSize
end
```

All class definition blocks (`classdef`, `properties`, `methods`, and `events`) support specific attributes. All attributes have default values. Specify attribute values only in cases where you want to change from the default value.

Note Specify the value of a particular attribute only once in any component block.

Attribute Descriptions

For lists of supported attributes, see:

- “Class Attributes” on page 6-5
- “Property Attributes” on page 8-8
- “Method Attributes” on page 9-4
- “Event Attributes” on page 11-16

Attribute Values

When you specify attribute values, those values affect all the components defined within the defining block. Defining properties with different attribute settings requires multiple properties blocks. Specify multiple attributes in a comma-separated list.

```
properties (SetObservable = true)
   AccountBalance
end

properties (SetAccess = private, Hidden = true)
   SSNumber
   CreditCardNumber
end
```

Simpler Syntax for true/false Attributes

You can use a simpler syntax for attributes whose values are `true` or `false`. The attribute name alone implies true and adding the not operator (`~`) to the name implies false. For example, these two ways of defining a static methods block are equivalent.

```
methods (Static)
   ...
end

methods (Static = true)
   ...
end
```

Similarly, these three ways of defining a nonstatic methods block are equivalent. All attributes that take a logical value have a default value of `false`, so you can omit the attribute to get the default behavior.

```
methods
   ...
end

methods (~Static)
   ...
end

methods (Static = false)
   ...
end
```

Enumeration Classes

Enumeration classes are specialized classes that define a fixed set of names representing a single type of value. Enumeration classes use an `enumeration` block that contains the enumeration members defined by the class.

The enumeration block starts with the `enumeration` keyword and terminates with the `end` keyword.

```
classdef ClassName < SuperClass
   enumeration
      EnumerationMember
   end
```

```
   ...
end
```

For example, this class defines two enumeration members that represent the logical values `false` and `true`.

```
classdef Boolean < logical
   enumeration
      No  (0)
      Yes (1)
   end
end
```

See “Define Enumeration Classes” on page 14-4 for more information.

Related Information

“Creating a Simple Class” on page 2-2

“Folders Containing Class Definitions” on page 6-14

Method Syntax

In this section...

This topic describes how to define class methods in MATLAB using `methods...end` blocks, and it introduces method argument validation. This topic focuses on nonstatic, concrete methods, also referred to as ordinary methods. For other types of methods, see:

- “Class Constructor Methods” on page 9-15
- “Handle Class Destructor” on page 7-13
- “Static Methods” on page 9-23
- “Property Get and Set Methods” on page 8-38
- “Abstract Classes and Class Members” on page 12-68

Methods Definition Block

The `methods` and `end` keywords define one or more class methods that have the same attribute settings. The methods themselves are defined using MATLAB function blocks. The syntax for defining a block of ordinary methods is:

```
methods (attributes)
   function method1(obj,arg1,...)
      ...
   end
   function method2(obj,arg1,...)
      ...
   end
   ...
end
```

With the exception of static methods, you must pass an object of the class explicitly to a MATLAB method.

For example, this class defines one public property and two public methods. Each method takes two input arguments: the object itself and a user-provided argument `inputArg`. The methods calculate the product and quotient of the value of the class property, `Property1`, and the input argument.

```
classdef methodDemo
   properties
      Property1
   end
   methods
      function prod = propMultiply(obj,inputArg)
         prod = obj.Property1*inputArg;
      end
      function quotient = propDivide(obj,inputArg)
         quotient = obj.Property1/inputArg;
      end
```

```
   end
end
```

You can also define multiple method blocks with different attributes. In this example, the first method is protected, and the second method is private. For more information, see “Method Attributes” on page 9-4.

```
classdef attributeDemo
   methods (Access = protected)
      function out = method1(obj,inputArg)
         ...
      end
   end
   methods (Access = private)
      function out = method2(obj,inputArg)
          ...
      end
   end
end
```

Method Argument Validation

You can define restrictions for method input and output arguments. To validate method arguments, add `arguments` blocks to your methods in the same way you would with a function. See `arguments` for more information.

Input Argument Validation

Input argument validation enables you to restrict the size, class, and other characteristics of a method input argument. The syntax for input argument validation is:

```
arguments
   argName1 (dimensions) class {validators} = defaultValue
   ...
end
```

- *`(dimensions)`* — Input size, specified as a comma-separated list of two or more numbers, such as `(1,2)`.
- *`class`* — Class, specified as a class name, such as `double` or the name of a user-defined class.
- *`{validators}`* — Comma-separated list of validation functions, such as `mustBeNumeric` and `mustBeScalarOrEmpty`, enclosed in curly brackets. For a list of validation functions, see “Argument Validation Functions”.
- *`defaultValue`* — Default values must conform to the specified size, type, and validation rules.

See `arguments` for the full description of elements in input argument validation syntax.

Input argument validation is useful for methods with public access. Restricting the types of argument values allowed from callers of the method can prevent errors before the body of the method is executed. For example, the `Rectangle` class represents a rectangle in the coordinate plane, with properties specifying its location (`X` and `Y`) as well as its width and height.

```
classdef Rectangle
    properties
        X (1,1) double {mustBeReal} = 0
        Y (1,1) double {mustBeReal} = 0
```

```
        Width (1,1) double {mustBeReal} = 0
        Height (1,1) double {mustBeReal} = 0
    end

    methods
        function R = enlarge(R,x,y)
            arguments (Input)
                R (1,1) Rectangle
                x (1,1) {mustBeNonnegative}
                y (1,1) {mustBeNonnegative}
            end
            arguments (Output)
                R (1,1) Rectangle
            end
            R.Width = R.Width + x;
            R.Height = R.Height + y;
        end
    end
end
```

The `enlarge` method increases the height and width of the rectangle by adding the user inputs `x` and `y` to `Width` and `Height`, respectively. Because the intent of the method is to enlarge one rectangle, the validation restricts the input arguments to scalar values with a `(1,1)` size restriction and nonnegative numeric values with `mustBeNonnegative`.

Instantiate the class, and call `enlarge` with inputs `5` and `-1` to `enlarge`. The argument validation returns an error.

```
rect1 = Rectangle;
rect1.enlarge(5,-1))
```

```
Error using Rectangle/enlarge
 rect1.enlarge(5,-1)
               ↑
Invalid argument at position 2. Value must be nonnegative.
```

Tip The `(Input)` attribute for an input arguments block is optional, but it is recommended for readability when defining both input and output argument blocks in a single method. MATLAB interprets an argument block with neither attribute as an input argument block.

Output Argument Validation

The syntax for output argument validation is the same as input validation, except that you must specify `(Output)` as an attribute of the arguments block, and you cannot set a default value.

```
arguments (Output)
   argName1 (dimensions) class {validators}
   ...
end
```

See `arguments` for the full description of the output argument validation syntax.

Output argument validation enables class authors to document what type of outputs a method returns, as well as acting as a fail-safe against future changes to the code that might alter the output types. For example, the `enlarge` method of `Rectangle` uses output argument validation to ensure that the object method returns a scalar instance of `Rectangle`.

```
arguments (Output)
   R (1,1) Rectangle
end
```

If `enlarge` were later revised to return only the dimensions of the `Rectangle` and not the object itself, for example, the output validation would help catch this potential mistake.

Special Considerations for Validation in Methods

Argument validation for class methods works much like it does for functions, but some aspects of argument validation are unique to methods.

- If a `classdef` file includes method prototypes for methods defined in separate files, any `arguments` blocks you want to define for those methods must be defined in the separate files. For more information on defining methods in separate files, see “Methods in Separate Files” on page 9-8.
- Subclass methods do not inherit argument validation. To preserve argument validation in subclass methods that override superclass methods, repeat the argument validation from the superclass method in the subclass method.
- Abstract methods do not support argument validation because they cannot define `arguments` blocks. For more information, see “Abstract Classes and Class Members” on page 12-68.
- Handle class destructor methods cannot use argument validation. In handle classes, a destructor method named `delete` is called by MATLAB when a handle object becomes unreachable or is explicitly deleted with a call to `delete`. A destructor must have exactly one input, no outputs, and no argument validation. MATLAB treats a method defined in any other way as an ordinary method and does not use it as a destructor. For more information, see “Handle Class Destructor” on page 7-13.

See Also

`arguments`

Related Examples

- “Function Argument Validation”

Call Superclass Methods on Subclass Objects

In this section...

Superclass Relation to Subclass

Subclasses can override superclass methods to support the greater specialization defined by the subclass. Because of the relationship that a subclass object is a superclass object, it is often useful to call the superclass version of the method before executing the specialized subclass code.

How to Call Superclass Methods

Subclass methods can call superclass methods if both methods have the same name. From the subclass, reference the method name and superclass name with the `@` symbol.

This is the syntax for calling a `superMethod` defined by `MySuperClass`:

```
superMethod@MySuperClass(obj,superMethodArguments)
```

For example, a subclass can call a superclass `disp` method to implement the display of the superclass part of the object. Then the subclass adds code to display the subclass part of the object:

```
classdef MySub < MySuperClass
   methods
      function disp(obj)
         disp@MySuperClass(obj)
            ...
      end
   end
end
```

How to Call Superclass Constructor

If you create a subclass object, MATLAB calls the superclass constructor to initialize the superclass part of the subclass object. By default, MATLAB calls the superclass constructor without arguments. If you want the superclass constructor called with specific arguments, explicitly call the superclass constructor from the subclass constructor. The call to the superclass constructor must come before any other references to the object.

The syntax for calling the superclass constructor uses an `@` symbol:

```
obj = obj@MySuperClass(SuperClassArguments)
```

In this class, the `MySub` object is initialized by the `MySuperClass` constructor. The superclass constructor constructs the `MySuperClass` part of the object using the specified arguments.

```
classdef MySub < MySuperClass
   methods
      function obj = MySub(arg1,arg2,...)
```

```
            obj = obj@MySuperClass(SuperClassArguments);
               ...
        end
    end
end
```

See “Subclass Constructors” on page 9-18 for more information.

See Also

Related Examples

- “Modify Inherited Methods” on page 12-13

Using a Class to Display Graphics

In this section...

Class Calculates Area

The `CircleArea` class shows the syntax of a typical class definition. This class stores a value for the radius of a circle and calculates the area of the circle when you request this information. `CircleArea` also implements methods to graph, display, and create objects of the class.

To use the `CircleArea` class, copy this code into a file named `CircleArea.m` and save this file in a folder that is on the MATLAB path.

```
classdef CircleArea
   properties
      Radius
   end
   properties (Constant)
      P = pi
   end
   properties (Dependent)
      Area
   end
   methods
      function obj = CircleArea(r)
         if nargin > 0
            obj.Radius = r;
         end
      end
      function val = get.Area(obj)
         val = obj.P*obj.Radius^2;
      end
      function obj = set.Radius(obj,val)
         if val < 0
            error('Radius must be positive')
         end
         obj.Radius = val;
      end
      function plot(obj)
         r = obj.Radius;
         d = r*2;
         pos = [0 0 d d];
         curv = [1 1];
         rectangle('Position',pos,'Curvature',curv,...
            'FaceColor',[.9 .9 .9])
         line([0,r],[r,r])
         text(r/2,r+.5,['r = ',num2str(r)])
         title(['Area = ',num2str(obj.Area)])
         axis equal
      end
      function disp(obj)
         rad = obj.Radius;
         disp(['Circle with radius: ',num2str(rad)])
```

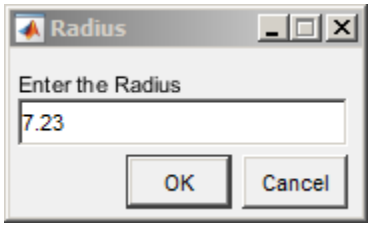
```
      end
   end
   methods (Static)
      function obj = createObj
         prompt = {'Enter the Radius'};
         dlgTitle = 'Radius';
         rad = inputdlg(prompt,dlgTitle);
         r = str2double(rad{:});
         obj = CircleArea(r);
      end
   end
end
```

Use the CircleArea Class

Create an object using the dialog box:

```
ca = CircleArea.createObj
```

Add a value for radius and click **OK**.



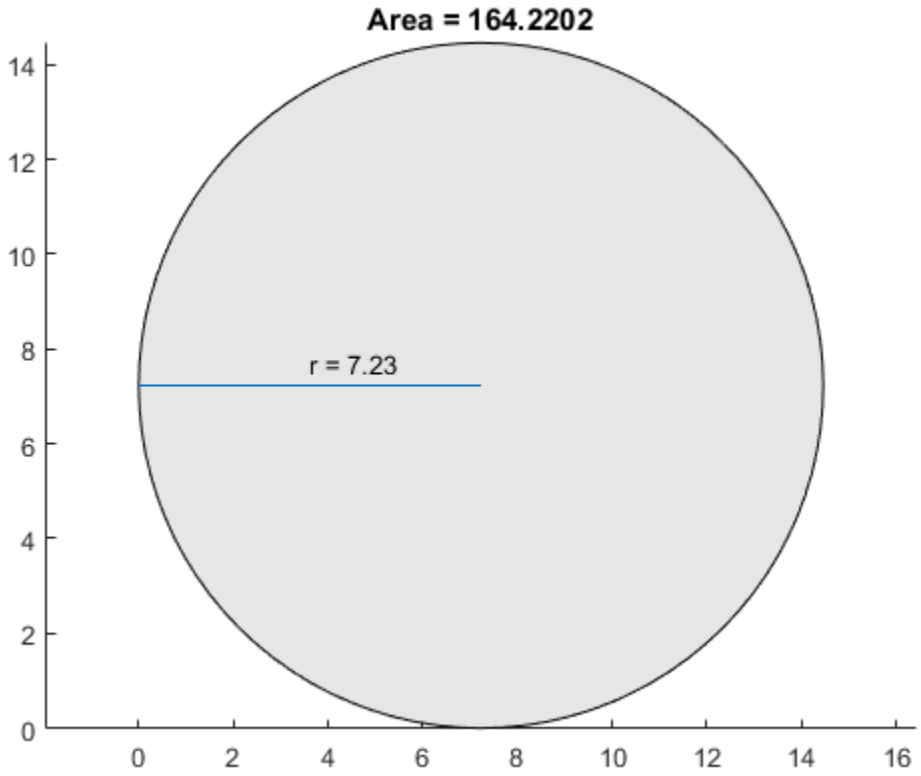
Query the area of the defined circle:

```
ca.Area
```

```
ans =

  164.2202
```

Call the overloaded plot method:

```
plot(ca)
```



Description of Class Definition

Class definition code begins with the `classdef` keyword followed by the class name:

```
classdef CircleArea
```

Define the `Radius` property within the `properties`-`end` keywords. Use default attributes:

```
   properties
      Radius
   end
```

Define the `P` property as `Constant` (“Define Class Properties with Constant Values” on page 15-2). Call the `pi` function only once when class is initialized.

```
   properties (Constant)
      P = pi
   end
```

Define the `Area` property as `Dependent` because its value depends on the `Radius` property.

```
   properties (Dependent)
      Area
   end
```

The `CircleArea` class constructor method has the same name as the class and accepts the value of the circle radius as an argument. This method also allows no input arguments. (“Class Constructor Methods” on page 9-15)

```
   methods
      function obj = CircleArea(r)
         if nargin > 0
            obj.Radius = r;
         else
            obj.Radius = 0;
         end
      end
```

Because the `Area` property is `Dependent`, the class does not store its value. The `get.Area` method calculates the value of the `Area` property whenever it is queried. (“Get and Set Methods for Dependent Properties” on page 8-42)

```
      function val = get.Area(obj)
         val = obj.P*obj.Radius^2;
      end
```

The `set.Radius` method tests the value assigned to the `Radius` property to ensure that the value is not less than zero. MATLAB calls `set.Radius` to assign a value to Radius. (“Property Get and Set Methods” on page 8-38)

```
      function obj = set.Radius(obj,val)
         if val < 0
            error('Radius must be positive')
         end
         obj.Radius = val;
      end
```

The `CircleArea` class overloads the `plot` function. The `plot` method uses the `rectangle` function to create a circle and draws the radius. (“Overload Functions in Class Definitions” on page 9-25)

```
      function plot(obj)
         r = obj.Radius;
         d = r*2;
         pos = [0 0 d d];
         curv = [1 1];
         rectangle('Position',pos,'Curvature',curv)
         line([0,r],[r,r])
         text(r/2,r+.5,['r = ',num2str(r)])
         axis equal
      end
```

The `CircleArea` class overloads the `disp` function to change the way MATLAB displays objects in the command window.

```
      function disp(obj)
         rad = obj.Radius;
         disp(['Circle with radius: ',num2str(rad)])
      end

end
methods (Static)
```

The `CircleArea` class defines a `Static` method that uses a dialog box to create an object. (“Static Methods” on page 9-23)

```
      function obj = createObj
         prompt = {'Enter the Radius'};
         dlgTitle = 'Radius';
         rad = inputdlg(prompt,dlgTitle);
         r = str2double(rad{:});
         obj = CircleArea(r);
      end
```

End of `Static methods` block and end of `classdef` block.

```
    end
end
```

MATLAB Code Analyzer Warnings

In this section...

Syntax Warnings and Property Names

The MATLAB Code Analyzer helps you optimize your code and avoid syntax errors while you write code. It is useful to understand some of the rules that the Code Analyzer applies in its analysis of class definition code. This understanding helps you avoid situations in which MATLAB allows code that is undesirable.

Variable/Property Name Conflict Warnings

The Code Analyzer warns about the use of variable names in methods that match the names of properties. For example, suppose that a class defines a property called `EmployeeName` and in this class, there is a method that uses `EmployeeName` as a variable:

```
properties
   EmployeeName
end
methods
   function someMethod(obj,n)
      EmployeeName = n;
   end
end
```

While the previous function is legal MATLAB code, it results in Code Analyzer warnings for two reasons:

- The value of `EmployeeName` is never used.
- `EmployeeName` is the name of a property that is used as a variable.

If the function `someMethod` contained the following statement instead:

```
obj.EmployeeName = n;
```

The Code Analyzer generates no warnings.

If you change `someMethod` to:

```
function EN = someMethod(obj)
   EN = EmployeeName;
end
```

The Code Analyzer returns only one warning, suggesting that you might actually want to refer to the `EmployeeName` property.

While this version of `someMethod` is legal MATLAB code, it is confusing to give a property the same name as a function. Therefore, the Code Analyzer provides a warning suggesting that you might have intended the statement to be:

```
EN = obj.EmployeeName;
```

Exception to Variable/Property Name Rule

Suppose that you define a method that returns a value of a property and uses the name of the property for the output variable name. For example:

```
function EmployeeName = someMethod(obj)
   EmployeeName = obj.EmployeeName;
end
```

The Code Analyzer does not warn when a variable name is the same as a property name when the variable is:

- An input or output variable
- A global or persistent variable

In these particular cases, the Code Analyzer does not warn you that you are using a variable name that is also a property name. Therefore, a coding error like the following:

```
function EmployeeName = someMethod(obj)
   EmployeeName = EmployeeName; % Forgot to include obj.
end
```

does not trigger a warning from the Code Analyzer.

See Also

Related Examples

- “Use of Editor and Debugger with Classes” on page 5-25

Objects in Conditional Statements

In this section...

Enable Use of Objects in Conditional Statements

Enable the use of objects in conditional statements by defining relational operators for the class of the object. Classes that derive from the `handle` class inherit relational operators. Value classes can implement operators to support the use of conditional statements involving objects. For information on defining operators for your class, see “Operator Overloading” on page 17-19.

How MATLAB Evaluates Switch Statements

MATLAB enables you to use objects in `switch` statements when the object’s class defines an `eq` method. The `eq` method implements the `==` operation on objects of that class.

For objects, *`case_expression == switch_expression`* defines how MATLAB evaluates `switch` and `cases` statements.

The values returned by the `eq` method must be of type `logical` or convertible to `logical`. MATLAB attempts to convert the output of `eq` to a logical value if the output of the `eq` method is a nonlogical value.

Note You do not need to define `eq` methods for enumeration classes. See “Enumerations in Switch Statements” on page 5-23.

Handle Objects in Switch Statements

All classes derived from the `handle` class inherit an `eq` method. The expression,

```
h1 == h2
```

is `true` if `h1` and `h2` are handles for the same object.

For example, the `BasicHandle` class derives from `handle`:

```
classdef BasicHandle < handle
   properties
      Prop1
   end
   methods
      function obj = BasicHandle(val)
         if nargin > 0
            obj.Prop1 = val;
         end
      end
```

```
   end
end
```

Create a `BasicHandle` object and use it in a `switch` statement:

```
h1 = BasicHandle('Handle Object');
h2 = h1;
```

Here is the `switch` statement code:

```
switch h1
   case h2
      disp('h2 is selected')
   otherwise
      disp('h2 not selected')
end
```

The result is:

```
h2 is selected
```

Object Must Be Scalar

The `switch` statements work only with scalar objects. For example:

```
h1(1) = BasicHandle('Handle Object');
h1(2) = BasicHandle('Handle Object');
h1(3) = BasicHandle('Handle Object');
h2 = h1;
```

```
switch h1
   case h2
      disp('h2 is selected')
   otherwise
      disp('h2 not selected')
end
```

The result is an error message.

```
SWITCH expression must be a scalar or string constant.
```

In this case, `h1` is not scalar. Use `isscalar` to determine if an object is scalar before entering a `switch` statement.

How to Define the eq Method

To enable the use of value-class objects in `switch` statements, implement an `eq` method for the class. Use the `eq` method to determine what constitutes equality of two objects of the class.

Behave like a Built-in Type

Some MATLAB functions also use the built-in == operator in their implementation. Therefore, your implementation of `eq` should be replaceable with the built-in `eq` to enable objects of your class work like built-in types in MATLAB code.

Design of eq

Implement the `eq` method to return a logical array representing the result of the == comparison.

For example, the `SwitchOnVer` class implements an `eq` method that returns `true` for the `==` operation if the value of the `Version` property is the same for both objects. In addition, `eq` works with arrays the same way as the built-in `eq`. For the following expression:

```
obj1 == obj2
```

The `eq` method works as follows:

- If both `obj1` and `obj2` are scalar, `eq` returns a scalar value.
- If both `obj1` and `obj2` are nonscalar arrays, then these arrays must have the same dimensions, and `eq` returns an array of the same size.
- If one input argument is scalar and the other is a nonscalar array, then `eq` treats the scalar object as if it is an array having the same dimensions as the nonscalar array.

Implementation of eq

Here is a class that implements an `eq` method. Ensure that your implementation contains appropriate error checking for the intended use.

```
classdef SwitchOnVer
   properties
      Version
   end
   methods
      function obj = SwitchOnVer(ver)
         if nargin > 0
            obj.Version = ver;
         end
      end
      function bol = eq(obj1,obj2)
         if ~strcmp(class(obj1),class(obj2))
            error('Objects are not of the same class')
         end
         s1 = numel(obj1);
         s2 = numel(obj2);
         if s1 == s2
            bol = false(size(obj1));
            for k=1:s1
               if obj1(k).Version == obj2(k).Version
                  bol(k) = true;
               else
                  bol(k) = false;
               end
            end
         elseif s1 == 1
            bol = scalarExpEq(obj2,obj1);
         elseif s2 == 1
            bol = scalarExpEq(obj1,obj2);
         else
            error('Dimension missmatch')
         end
         function ret = scalarExpEq(ns,s)
            % ns is nonscalar array
            % s is scalar array
            ret = false(size(ns));
            n = numel(ns);
            for kk=1:n
```

```
                if ns(kk).Version == s.Version
                   ret(kk) = true;
                else
                   ret(kk) = false;
                end
             end
          end
       end
    end
end
```

Use `SwitchOnVer` objects in `switch` statements:

```
% Create known versions of objects
ov1 = SwitchOnVer(1.0);
ov2 = SwitchOnVer(2.0);
ov3 = SwitchOnVer(3.0);
...

...
if isscalar(objIn)
       switch(objIn)
          case ov1
             disp('This is version 1.0')
          case ov2
             disp('This is version 2.0')
          case ov3
             disp('This is version 3.0')
          otherwise
             disp('There is no version')
       end
    else
       error('Input object must be scalar')
    end
```

Enumerations in Switch Statements

MATLAB enables you to use enumerations in `switch` statements without requiring an explicitly defined `eq` method for the enumeration class.

For example, the `WeeklyPlanner` class defines enumerations for five days of the week. The `switch`/`case` statements in the `todaySchedule` static method dispatch on the enumeration member corresponding to the current day of the week. The `datetime`, `day`, and `string` functions return a string containing the name of the current day.

```
classdef WeeklyPlanner
   enumeration
      Monday, Tuesday, Wednesday, Thursday, Friday
   end
   methods (Static)
      function todaySchedule
         dayName = string(day(datetime("name"));
         dayEnum =  WeeklyPlanner.(dayName);
         switch dayEnum
            case WeeklyPlanner.Monday
               disp("Monday schedule")
            case WeeklyPlanner.Tuesday
```

```
            disp("Tuesday schedule")
         case WeeklyPlanner.Wednesday
            disp("Wednesday schedule")
         case WeeklyPlanner.Thursday
            disp("Thursday schedule")
         case WeeklyPlanner.Friday
            disp("Friday schedule")
      end
   end
end
```

Call `todaySchedule` to display today's schedule:

```
WeeklyPlanner.todaySchedule
```

Enumerations Derived from Built-In Types

Enumeration classes that derived from built-in types inherit the superclass `eq` method. For example, the `FlowRate` class derives from `int32`:

```
classdef FlowRate < int32
   enumeration
      Low    (10)
      Medium (50)
      High   (100)
   end
end
```

The `switchEnum` function switches on the input argument, which can be a `FlowRate` enumeration value.

```
function switchEnum(inpt)
   switch inpt
      case 10
         disp('Flow = 10 cfm')
      case 50
         disp('Flow = 50 cfm')
      case 100
         disp('Flow = 100 cfm')
   end
end
```

Call `switchEnum` with an enumerated value:

```
switchEnum(FlowRate.Medium)
```

```
Flow = 50 cfm
```

Use of Editor and Debugger with Classes

In this section...

Write Class Code in the Editor

The MATLAB code editor provides an effective environment for class development. The Code Analyzer, which is built into the editor, check code for problems and provides information on fixing these problems. For information on editor use and features, see `edit`.

How to Refer to Class Files

Define classes in files just like scripts and functions. To use the editor or debugger with a class file, use the full class name. For example, suppose the file for a class, `myclass.m` is in the following location:

```
+PackFld1/+PackFld2/@myclass/myclass.m
```

To open `myclass.m` in the MATLAB editor, you could reference the file using dot-separated package names:

```
edit PackFld1.PackFld2.myclass
```

You could also use path notation:

```
edit +PackFld1/+PackFld2/@myclass/myclass
```

If `myclass.m` is not in a class folder, then enter:

```
edit +PackFld1/+PackFld2/myclass
```

To refer to functions inside a package folder, use dot or path separators:

```
edit PackFld1.PackFld2.packFunction
edit +PackFld1/+PackFld2/packFunction
```

To refer to a method defined in its own file inside a class folder, use:

```
edit +PackFld1/+PackFld2/@myclass/myMethod
```

How to Debug Class Files

For debugging, `dbstop` enables you to set breakpoints in the class constructor by specifying the fully qualified class file name. To set a breakpoint at a method defined in the class file, specify the line number of the method with the `dbstop` command. For example, if the method begins on line 14 in the `classdef` file, `myclass.m`, use this command to put a breakpoint on the first executable line of the method.

```
dbstop in myclass at 14
```

See “Automatic Updates for Modified Classes” on page 5-27 for information about clearing class after modification.

See Also

`dbstop`

Related Examples

- “MATLAB Code Analyzer Warnings” on page 5-18
- “Debug MATLAB Code Files”

Automatic Updates for Modified Classes

In this section...

When MATLAB Loads Class Definitions

MATLAB loads a class definition:

- The first time the class is referenced, such as creating an instance, accessing a constant property, or calling a static method of the class.
- Whenever the definition of a loaded class changes and MATLAB returns to the command prompt.
- When you change the MATLAB path and cause a different definition of the class to be used. The change takes effect after MATLAB returns to the command prompt.
- Whenever you access the class metadata.

MATLAB allows only one definition for a class to exist at any time. Therefore, MATLAB attempts to update all existing objects of a class automatically to conform to the new class definition. You do not need to call `clear classes` to remove existing objects when you change their defining class.

Note Using an editor other than the MATLAB editor or using MATLAB Online™ can result in delays to automatic updating.

Consequences of Automatic Update

MATLAB follows a set of basic rules when updating existing objects. An automatic update can result in:

- Existing objects being updated to the new class definition.
- An error if MATLAB cannot convert the objects to the new class definition or if there is an error in the class definition itself.

Here is an example of what happens when you create an instance of a concrete class edit the class definition to make the class abstract.

```
a = MyClass;
% Edit MyClass to make it Abstract

a

Error using MyClass/display
Cannot update object because the class 'MyClass' is now abstract.
```

Note MATLAB does not update metaclass instances when you change the definition of a class. You must get new metaclass data after updating a class definition.

What Happens When Class Definitions Change

MATLAB updates existing objects when a class definition changes, including the following situations:

- Value change to handle — Existing objects become independent handles referring to different objects.
- Enumeration member added — Existing objects preserve the enumeration members they had previously, even if the underlying values have changed.
- Enumeration member removed — Existing objects that are not using the removed member have the same enumeration members that they had previously. Existing objects that use the removed member replace the removed member with the default member of the enumeration.
- Enumeration block removed — Enumeration members are taken out of use.
- Superclass definition changed — Changes applied to all subclasses in the hierarchy of that superclass.
- Superclass added or removed — Change of superclass applied to all existing objects.

Ensure Defining Folder Remains in Scope

Changes to the MATLAB path that result in removing the class definition file from the path, even temporarily, can produce side effects. If a function changes from the current folder, which contains the class definition, and that folder is not on the path, then the function cannot call methods of the class that is now out of scope. To avoid potential problems, add the class defining folder to the path before changing to another folder.

For example, suppose the class of the input `obj` is defined in the current folder, which is not on the path. Before changing the current folder to another folder, add the current folder to the path using the `addpath` function.

```
function runFromTempFolder(obj)
    % Add current folder to path
    addpath(pwd)
    definingFolder = cd('myTempFolder');
    obj.myMethod;
    cd(definingFolder)
end
```

Actions That Do Not Trigger Updates

These actions do not update existing objects:

- Calling the `class` function on an out-of-date object
- Assigning an out-of-date object to a variable
- Calling a method that does not access class data
- Changing property validation in the class definition (“Validate Property Values” on page 8-18)

Objects do not update until referenced in a way that exposes the change, such as invoking the object display or assigning to a property.

Multiple Updates to Class Definitions

Updates do not occur incrementally. Updates conform to the latest version of the class.

Object Validity with Deleted Class File

Deleting a class definition file does not make instances of that class invalid. However, you cannot call methods on existing objects of that class.

When Updates Are Not Possible

Some class updates result in an invalid class definition. In these cases, objects do not update until the error is resolved:

- Adding a superclass can result in a property or method being defined twice.
- Changing a superclass to be `Sealed` when objects of one of its subclasses exists results in an invalid subclass definition.

Some class updates cause situations in which MATLAB cannot update existing objects to conform to a modified class definition. These cases result in errors until you delete the objects:

- Adding an enumeration block to a non-enumeration class
- Redefining a class to be abstract
- Removing a class from a heterogeneous hierarchy that results in there being no default object to replace existing objects in a heterogeneous array
- Updating a class to restrict array formation behavior, such as overloading array indexing and concatenation.
- Inheriting a `subsref`, `subsasgn`, `cat`, `vertcat`, or `horzcat` method
- Redefining a handle class to be a value class.

Potential Consequences of Class Updates

- Following an update, existing objects can be incompatible with the new class definition. For example, a newly added property can require execution of the constructor to be valid.
- Removing or renaming properties can lose the data held in the property. For example, if a property holds the only reference to another object and you remove that property from the class, the MATLAB deletes the object because there are no longer any references to it.

- Removing a class from a heterogeneous class hierarchy can result in invalid heterogeneous array elements. In this case, the default object for the heterogeneous hierarchy replaces these array elements.

Interactions with the Debugger

Since R2021a.

MATLAB disables the debugger during class updates. Before R2021a, a breakpoint could potentially interrupt the class update process and allow for the introduction of errors when the update resumes. For example, the breakpoint allows a class author to introduce invalid syntax into the class definition or remove the class from the path entirely, potentially causing MATLAB to crash.

Example of a Disabled Breakpoint

This class defines a property validation function:

```
classdef ClassWithBreakpoint
    properties (Constant)
        Prop1 (1,1) {myPropertyValidator} = 32
    end
end

function myPropertyValidator(~)
end % Add breakpoint here
```

Create an instance of this class. Then add a breakpoint where indicated, and update the definition of `Prop1` to a different initial value:

```
Prop1 (1,1) {myPropertyValidator} = 10
```

In version R2020b and earlier, MATLAB hits the breakpoint, and the class update is interrupted. In R2021a, the debugger is disabled, and the breakpoint does not interrupt the update.

Updates to Class Attributes

Changing class attributes can change existing object behavior or make the objects invalid. MATLAB returns an error when you access the invalid objects.

Change	Effect
Make `Abstract = true`	Accessing existing objects returns an error.
Change `AllowedSubclasses`	Newly created objects can inherit from different superclasses than existing objects.
Change `ConstructOnLoad`	Loading classes obeys the current value of `ConstructOnLoad`.
Change `HandleCompatible`	Newly created objects can have different class hierarchy than existing objects.
Change `Hidden`	Appearance of class in list of superclasses and access by `help` function can change
Change `InferiorClasses`	Method dispatching for existing objects can change.
Make `Sealed = true`	Existing subclass objects return errors when accessed.

Updates to Property Definitions

When you change the definition of class properties, MATLAB applies the changes to existing objects of the class.

Change	Effect
Add property	Adds the new property to existing objects of the class. Sets the property values to the default value (which is `[]` if the class definition does not specify a default).
Remove property	Removes the property from existing objects of the class. Attempts to access the removed property fail.
Change property default value	Does not apply the new default value to existing objects of the class.
Move property between subclass and superclass	Does not apply different default value when property definition moves between superclass and subclass.
Change property attribute value	Applies changes to existing objects of the class. Some cases require transitional steps: • `Abstract` — Existing objects of a class that becomes abstract cannot be updated. Delete these objects. • `Access` — Changes to the `public`, `protected`, or `private` property access settings affect access to existing objects. Changes to the access lists do not change existing objects. However, if you add classes to the access list, instances of those classes have access to this property. If you remove classes from the access list, objects of those classes no longer have access to this property. • `Dependent` — If changed to `true`, existing objects no longer store property values. If you want to query the property value, add a property get method for the property. • `Transient` — If changed to `true`, objects already saved, reload this property value. If changed to `false`, objects already saved reload this property using the default value.

Updates to Method Definitions

When you change the definition of class methods, MATLAB changes the affected class member in existing objects as follows.

Change	Effect
Add method	You can call the new method on existing objects of the class.
Modify method	Modifications are available to existing objects.
Remove method	You can on longer call deleted method on existing objects.

Change	Effect
Change method attribute value	Apply changes to existing objects of the class. Some cases require transitional steps: • `Abstract` — Existing objects of a class that becomes abstract cannot be updated. Delete these objects. • `Access` — Changes to method `public`, `protected`, or `private` access settings affect access to existing objects. Changes to the access lists do not change existing instances. However, if you add classes to the access list, instances of those classes have access to this method. If you remove classes from the access list, objects of those classes no longer have access to this method. • `Sealed` — If changed to `true` and existing subclasses already have defined the method, MATLAB returns an error because the new class definition cannot be applied to existing subclasses.

Updates to Event Definitions

Change	Effect
Add event	Existing objects of the class support the new event.
Change event name	New event name is visible to existing objects of the class. MATLAB: • Does not update existing metaclass objects • Does update newly acquired metaclass objects • Does not update listeners to use new event name
Remove event	Existing objects no longer support deleted event.

Change	Effect
Change event attribute value	Apply changes to existing objects of the class. Some cases require transitional steps: • `ListenAccess` — Changes to event `public`, `protected`, or `private` listen access settings affect access to existing objects. Changes to the access list do not change existing objects. However, if you add classes to the access list, objects of those classes can create listeners for this event. If you remove classes from the access list, objects of those classes are not allowed to create listeners for this event. • `NotifyAccess` — Changes to event `public`, `protected`, or `private` notify access settings affect access to existing objects. Changes to the access list do not change existing objects. However, if you add classes to the access list, instances of those classes can trigger this event. If you remove classes, objects of those classes are not able to trigger this event.

See Also

Related Examples

- “Use of Editor and Debugger with Classes” on page 5-25

Comparison of MATLAB and Other OO Languages

In this section...

Some Differences from C++ and Java Code

The MATLAB programming language differs from other object-oriented languages, such as C++ or Java in some important ways.

Public Properties

Unlike fields in C++ or the Java language, you can use MATLAB properties to define a public interface separate from the implementation of data storage. You can provide public access to properties because you can define set and get access methods that execute automatically when assigning or querying property values. For example, the following statement:

```
myobj.Material = 'plastic';
```

assigns the `char` vector `plastic` to the `Material` property of `myobj`. Before making the actual assignment, `myobj` executes a method called `set.Material` (assuming the class of `myobj` defines this method), which can perform any necessary operations. See “Property Get and Set Methods” on page 8-38 for more information on property access methods.

You can also control access to properties by setting attributes, which enable public, protected , or private access. See “Property Attributes” on page 8-8 for a full list of property attributes.

No Implicit Parameters

In some languages, one object parameter to a method is always implicit. In MATLAB, objects are explicit parameters to the methods that act on them.

Dispatching

In MATLAB classes, method dispatching is not based on method signature, as it is in C++ and Java code. When the argument list contains objects of equal precedence, MATLAB uses the leftmost object to select the method to call.

However, if the class of an argument is superior to the class of the other arguments, MATLAB dispatches to the method of the superior argument, regardless of its position within the argument list.

See “Class Precedence” on page 6-19 for more information.

Calling Superclass Method

- In C++, you call a superclass method using the scoping operator: *`superclass::method`*
- In Java code, you use: *`superclass.method`*

The equivalent MATLAB operation is *`method@superclass`*.

Other Differences

In MATLAB classes, there is no equivalent to C++ templates or Java generics. However, MATLAB is weakly typed and it is possible to write functions and classes that work with different types of data.

MATLAB classes do not support overloading functions using different signatures for the same function name.

Object Modification

MATLAB classes can define public properties, which you can modify by explicitly assigning values to those properties on a given instance of the class. However, only classes derived from the `handle` class exhibit reference behavior. Modifying a property value on an instance of a value classes (classes not derived from `handle`), changes the value only within the context in which the modification is made.

The sections that follow describe this behavior in more detail.

Objects Passed to Functions

MATLAB passes all variables by value. When you pass an object to a function, MATLAB copies the value from the caller into the parameter variable in the called function.

However, MATLAB supports two kinds of classes that behave differently when copied:

- Handle classes — a handle class instance variable refers to an object. A copy of a handle class instance variable refers to the same object as the original variable. If a function modifies a handle object passed as an input argument, the modification affects the object referenced by both the original and copied handles.
- Value classes — the property data in an instance of a value class are independent of the property data in copies of that instance (although, a value class property could contain a handle). A function can modify a value object that is passed as an input argument, but this modification does not affect the original object.

See “Comparison of Handle and Value Classes” on page 7-2 for more information on the behavior and use of both kinds of classes.

Passing Value Objects

When you pass a value object to a function, the function creates a local copy of the argument variable. The function can modify only the copy. If you want to modify the original object, return the modified object and assign it to the original variable name. For example, consider the value class, `SimpleClass` :

```
classdef SimpleClass
   properties
      Color
   end
   methods
      function obj = SimpleClass(c)
         if nargin > 0
            obj.Color = c;
         end
```

```
      end
   end
end
```

Create an instance of `SimpleClass`, assigning a value of `red` to its `Color` property:

```
obj = SimpleClass('red');
```

Pass the object to the function `g`, which assigns `blue` to the `Color` property:

```
function y = g(x)
   x.Color = 'blue';
   y = x;
end

y = g(obj);
```

The function `g` modifies its copy of the input object and returns that copy, but does not change the original object.

```
y.Color

ans =

    blue

obj.Color

ans =

     red
```

If the function `g` did not return a value, the modification of the object `Color` property would have occurred only on the copy of `obj` within the function workspace. This copy would have gone out of scope when the function execution ended.

Overwriting the original variable actually replaces it with a new object:

```
obj = g(obj);
```

Passing Handle Objects

When you pass a handle to a function, the function makes a copy of the handle variable, just like when passing a value object. However, because a copy of a handle object refers to the same object as the original handle, the function can modify the object without having to return the modified object.

For example, suppose that you modify the `SimpleClass` class definition to make a class derived from the `handle` class:

```
classdef SimpleHandleClass < handle
   properties
      Color
   end
   methods
      function obj = SimpleHandleClass(c)
         if nargin > 0
            obj.Color = c;
         end
      end
```

```
   end
end
```

Create an instance of `SimpleHandleClass`, assigning a value of `red` to its `Color` property:

```
obj = SimpleHandleClass('red');
```

Pass the object to the function `g`, which assigns `blue` to the `Color` property:

```
y = g(obj);
```

The function `g` sets the `Color` property of the object referred to by both the returned handle and the original handle:

```
y.Color
```

```
ans =

blue
```

```
obj.Color
```

```
ans =

blue
```

The variables `y` and `obj` refer to the same object:

```
y.Color = 'yellow';
obj.Color
```

```
ans =

yellow
```

The function `g` modified the object referred to by the input argument (`obj`) and returned a handle to that object in `y`.

MATLAB Passes Handles by Value

A handle variable is a reference to an object. MATLAB passes this reference by value.

Handles do not behave like references in C++. If you pass an object handle to a function and that function assigns a different object to that handle variable, the variable in the caller is not affected. For example, suppose you define a function `g2`:

```
function y = g2(x)
   x = SimpleHandleClass('green');
   y = x;
end
```

Pass a handle object to `g2`:

```
obj = SimpleHandleClass('red');
y = g2(obj);
y.Color
```

```
ans =

green
```

```
obj.Color

ans =

red
```

The function overwrites the handle passed in as an argument, but does not overwrite the object referred to by the handle. The original handle `obj` still references the original object.

Static Properties

In MATLAB, classes can define constant properties, but not "static" properties in the sense of other languages like C++. You cannot change constant properties from the initial value specified in the class definition.

MATLAB has long-standing rules that variables always take precedence over the names of functions and classes. Assignment statements introduce a variable if one does not exist.

Expressions of this form

```
A.B = C
```

Introduce a new variable, `A`, that is a `struct` containing a field `B` whose value is `C`. If `A.B = C` could refer to a static property of class `A`, then class `A` would take precedence over variable `A`.

This behavior would be a significant incompatibility with prior releases of MATLAB. For example, the introduction of a class named `A` on the MATLAB path could change the meaning of an assignment statement like `A.B = C` inside a `.m` code file.

In other languages, classes rarely use static data, except as private data within the class or as public constants. In MATLAB, you can use constant properties the same way you use `public final static` fields in Java. To use data that is internal to a class in MATLAB, create persistent variables in private or protected methods or local functions used privately by the class.

Avoid static data in MATLAB. If a class has static data, using the same class in multiple applications causes conflicts among applications. Conflicts are less of an issue in some other languages. These languages compile applications into executables that run in different processes. Each process has its own copy of the class static data. MATLAB, frequently runs many different applications in the same process and environment with a single copy of each class.

For ways to define and use static data in MATLAB, see “Static Data” on page 4-2.

Common Object-Oriented Techniques

This table provides links to sections that discuss object-oriented techniques commonly used by other object-oriented languages.

Technique	How to Use in MATLAB
Operator overloading	“Operator Overloading” on page 17-19
Multiple inheritance	“Subclassing Multiple Classes” on page 12-19
Subclassing	“Design Subclass Constructors” on page 12-7
Destructor	“Handle Class Destructor” on page 7-13

Technique	How to Use in MATLAB
Data member scoping	“Property Attributes” on page 8-8
Packages (scoping classes)	“Packages Create Namespaces” on page 6-21
Named constants	See “Define Class Properties with Constant Values” on page 15-2 and “Named Values” on page 14-2
Enumerations	“Define Enumeration Classes” on page 14-4
Static methods	“Static Methods” on page 9-23
Static properties	Not supported. See `persistent` variables. For the equivalent of Java `static final` or C++ `static const` properties, use `Constant` properties. See “Define Class Properties with Constant Values” on page 15-2 For mutable static data, see “Static Data” on page 4-2
Constructor	“Class Constructor Methods” on page 9-15
Copy constructor	No direct equivalent
Reference/reference classes	“Comparison of Handle and Value Classes” on page 7-2
Abstract class/Interface	“Abstract Classes and Class Members” on page 12-68
Garbage collection	“Object Lifecycle” on page 7-16
Instance properties	“Dynamic Properties — Adding Properties to an Instance” on page 8-47
Importing classes	“Import Classes” on page 6-25
Events and Listeners	“Event and Listener Concepts” on page 11-12

Defining and Organizing Classes

User-Defined Classes

In this section...

What Is a Class Definition

A MATLAB class definition is a template whose purpose is to provide a description of all the elements that are common to all instances of the class. Class members are the properties, methods, and events that define the class.

Define MATLAB classes in code blocks, with subblocks delineating the definitions of various class members. For syntax information on these blocks, see “Components of a Class” on page 5-2.

Attributes for Class Members

Attributes modify the behavior of classes and the members defined in the class-definition block. For example, you can specify that methods are static or that properties are private. The following sections describe these attributes:

- “Class Attributes” on page 6-5
- “Method Attributes” on page 9-4
- “Property Attributes” on page 8-8
- “Event Attributes” on page 11-16

Class definitions can provide information, such as inheritance relationships or the names of class members without actually constructing the class. See “Class Metadata” on page 16-2.

See “Specifying Attributes” on page 6-7 for more on attribute syntax.

Kinds of Classes

There are two kinds of MATLAB classes—handle classes and value classes.

- Value classes represent independent values. Value objects contain the object data and do not share this data with copies of the object. MATLAB numeric types are value classes. Values objects passed to and modified by functions must return a modified object to the caller.
- Handle classes create objects that reference the object data. Copies of the instance variable refer to the same object. Handle objects passed to and modified by functions affect the object in the caller’s workspace without returning the object.

For more information, see “Comparison of Handle and Value Classes” on page 7-2.

Constructing Objects

For information on class constructors, see “Class Constructor Methods” on page 9-15.

For information on creating arrays of objects, see “Construct Object Arrays” on page 10-2.

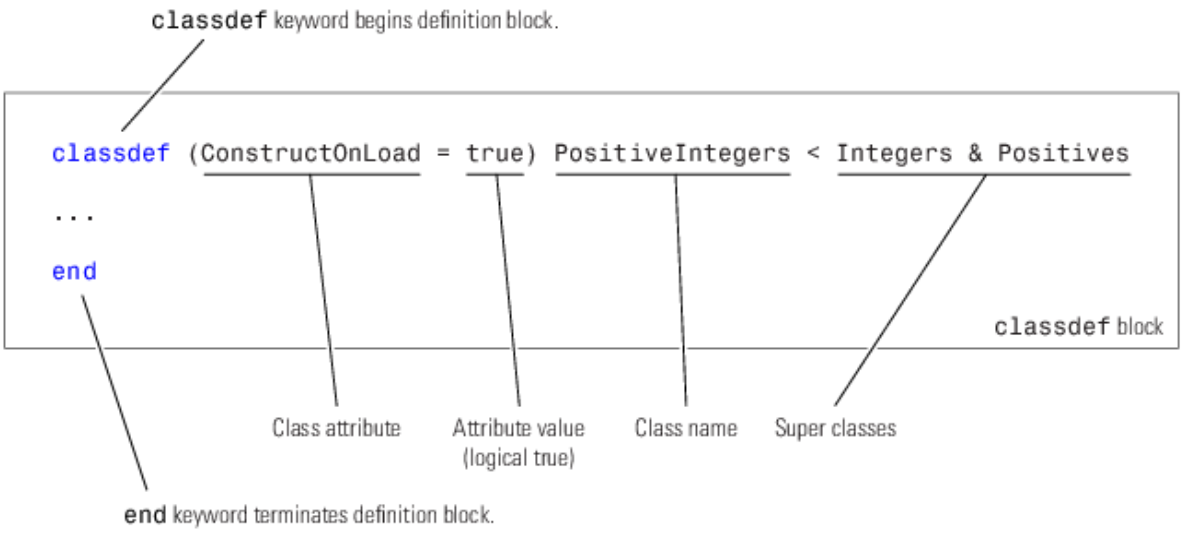
Class Hierarchies

For more information on how to define class hierarchies, see “Hierarchies of Classes — Concepts” on page 12-2.

classdef Syntax

Class definitions are blocks of code that are delineated by the `classdef` keyword at the beginning and the `end` keyword at the end. Files can contain only one class definition.

The following diagram shows the syntax of a `classdef` block. Only comments and blank lines can precede the `classdef` keyword.



Class Code

Here is a simple class definition with one property and a constructor method that sets the value of the property when there is an input argument supplied.

```
classdef MyClass
   properties
      Prop
   end
   methods
      function obj = MyClass(val)
         if nargin > 0
            obj.Prop = val;
         end
      end
```

```
    end
end
```

To create an object of `MyClass`, save the class definition in a `.m` file having the same name as the class and call the constructor with any necessary arguments:

```
d = datestr(now);
o = MyClass(d);
```

Use dot notation to access the property value:

```
o.Prop

ans =

10-Nov-2005 10:38:14
```

The constructor should support a no argument syntax so MATLAB can create default objects. For more information, see “No Input Argument Constructor Requirement” on page 9-18.

For more information on the components of a class definition, see “Components of a Class” on page 5-2

See Also

Related Examples

- “Creating a Simple Class” on page 2-2
- “Developing Classes That Work Together” on page 3-6
- “Representing Structured Data with Classes” on page 3-14

Class Attributes

In this section...

Specifying Class Attributes

All classes support the attributes listed in the following table. Attributes enable you to modify the behavior of class. Attribute values apply to the class defined within the `classdef` block.

```
classdef (Attribute1 = value1, Attribute2 = value2,...) ClassName
   ...
end
```

Class Attributes

Attribute Name	Class	Description
`Abstract`	`logical` (default = `false`)	If specified as `true`, this class is an abstract class (cannot be instantiated). See “Abstract Classes and Class Members” on page 12-68 for more information.
`AllowedSubclasses`	`meta.class` object or cell array of `meta.class` objects	List classes that can subclass this class. Specify subclasses as `meta.class` objects in the form: • A single `meta.class` object • A cell array of `meta.class` objects. An empty cell array, `{}`, is the same as a `Sealed` class (no subclasses). Specify `meta.class` objects using the `?`*`ClassName`* syntax only. See “Specify Allowed Subclasses” on page 12-21 for more information.
`ConstructOnLoad`	`logical` (default = `false`)	If `true`, MATLAB calls the class constructor when loading an object from a MAT-file. Classes defined with this attribute must have a no-argument constructor. See “Initialize Objects When Loading” on page 13-22 for more information.
`HandleCompatible`	`logical` (default = `false`) for value classes	If specified as `true`, this class can be used as a superclass for handle classes. All handle classes are `HandleCompatible` by definition. See “Handle Compatible Classes” on page 12-31 for more information.
`Hidden`	`logical` (default = `false`)	If `true`, this class does not appear in the output of the `superclasses` or `help` functions.
`InferiorClasses`	`meta.class` object or cell array of `meta.class` objects	Use this attribute to establish a precedence relationship among classes. Specify a cell array of `meta.class` objects using the `?` operator. The fundamental classes are always inferior to user-defined classes and do not show up in this list. See “Class Precedence” on page 6-19.
`Sealed`	`logical` (default = `false`)	If `true`, this class cannot be subclassed.
Framework attributes	Classes that use certain framework base classes have framework-specific attributes. See the documentation for the specific base class you are using for information on these attributes.	

Specifying Attributes

Attributes are specified for class members in the `classdef`, `properties`, `methods`, and `events` definition blocks. The particular attribute setting applies to all members defined within that particular block. You can use multiple `properties`, `methods`, and `events` definition blocks to apply different attribute setting to different class members.

Superclass Attribute Values Are Not Inherited

Class attributes settings are not inherited, so superclass attribute values do not affect subclasses.

Attribute Syntax

Specify class attribute values in parentheses, separating each attribute name/attribute value pair with a comma. The attribute list always follows the `classdef` or class member keyword, as shown:

```
classdef (attribute-name = expression, ...) ClassName

   properties (attribute-name = expression, ...)
      ...
   end
   methods (attribute-name = expression, ...)
      ...
   end
   events (attribute-name = expression, ...)
      ...
   end
end
```

Class-Specific Attributes

Some MATLAB classes define additional attributes that you can use only with the class hierarchies that define these attributes. See the specific documentation for the classes you are using for information on any additional attributes supported by those classes.

See Also

More About

- “Expressions in Attribute Specifications” on page 6-10

Functions Inside Class Definition Files

Just as you can define local functions in a script file or function file, you can also define local functions inside a `classdef` file. Define these functions outside of the `classdef` block, but in the same file as the class definition. You can call these functions from anywhere in the same file, but they are not visible outside of the file in which you define them.

Local functions in `classdef` files are useful for utility functions that you use only within that file. For example, this code defines `myUtilityFcn` outside the `classdef` block.

```
classdef MyClass
   properties
      PropName
   end
   methods
      function obj = method1(val)
         adjustedVal = myUtilityFcn(val)
         ...
      end
   end
end % End of classdef

function out = myUtilityFcn(in)
   ...
end
```

When you call `method1` of `MyClass`, the method first uses `myUtilityFcn` to perform some preprocessing on the input argument before performing any other actions.

Unlike methods, these functions do not require an instance of the class as an input, but they can take or return arguments that are instances of the class and access the members of those instances, including private members. However, if a function inside a class definition file needs direct access to class members, consider defining the function as a method of the class instead.

See Also

`classdef`

More About

- “Folders Containing Class Definitions” on page 6-14

Evaluation of Expressions in Class Definitions

In this section...

Why Use Expressions

An expression used in a class definition can be any valid MATLAB statement that evaluates to a single array. Use expressions to define property default values and in attribute specifications. Expressions are useful to derive values in terms of other values. For example, suppose that you want to define a constant property with the full precision value of 2π. You can assign the property the value returned by the expression `2*pi`. MATLAB evaluates the function when first loading the class.

For information on assigning property default values and attribute values, see “Initialize Property Values” on page 8-13 and “Property Attributes” on page 8-8.

Where to Use Expressions in Class Definitions

Here are some examples of expressions used in a class definition:

```
classdef MyClass
   % Some attributes are set to logical values
   properties (Constant = true)
      CnstProp = 2*pi
   end
   properties
      % Static method of this class
      Prop1 = MyClass.setupAccount
      % Constant property from this class
      Prop2 = MyClass.CnstProp
      % Function that returns a value
      Prop3 = datestr(now)
      % A class constructor
      Prop4 = AccountManager
   end
   methods (Static)
      function accNum = setupAccount
         accNum = randi(9,[1,12]);
      end
   end
end
```

MATLAB does not call property set methods when assigning the result of default value expressions to properties. (See “Property Get and Set Methods” on page 8-38 for information about these special methods.)

Enumerations that derived from MATLAB types can use expressions to assign a value:

```
classdef FlowRate < int32
   enumeration
      Low    (10)
      Medium (FlowRate.Low*5)
      High   (FlowRate.Low*10)
   end
end
```

MATLAB evaluates these expressions only once when enumeration members are first accessed.

Expressions in Attribute Specifications

For attributes values that are logical `true` or `false`, class definitions can specify attribute values using expressions. For example, this assignment makes `MyClass` sealed (cannot be subclassed) for versions of MATLAB before R2014b (`verLessThan`)

```
classdef  (Sealed = verLessThan('matlab','8.4')) MyClass
```

The expression on the right side of the equal sign (=) must evaluate to `true` or `false`. You cannot use any definitions from the class file in this expression, including any constant properties, static methods, and local functions.

While you can use conditional expression to set attribute values, doing so can cause the class definition to change based on external conditions. Ensure that this behavior is consistent with your class design.

Note The `AllowedSubclasses` and the `InferiorClasses` attributes require an explicit specification of a cell array of `meta.class` objects as their values. You cannot use expressions to return these values.

Expressions That Specify Default Property Values

Property definitions allow you to specify default values for properties using any expression that has no reference to variables. For example, `VectorAngle` defines a constant property (`Rad2Deg`) and uses it in an expression that defines the default value of another property (`Angle`). The default value expression also uses a static method (`getAngle`) defined by the class:

```
classdef VectorAngle
   properties (Constant)
      Rad2Deg = 180/pi
   end
   properties
      Angle = VectorAngle.Rad2Deg*VectorAngle.getAngle([1 0],[0 1])
   end
   methods
      function obj = VectorAngle(vx,vy)
         obj.Angle = VectorAngle.getAngle(vx,vy);
      end
   end
   methods (Static)
      function r = getAngle(vx,vy)
         % Calculate angle between 2D vectors
         cr = vx(1)*vy(1) + vx(2)*vy(2)/sqrt(vx(1)^2 + vx(2)^2) * ...
            sqrt(vy(1)^2 + vy(2)^2);
```

```
            r = acos(cr);
        end
    end
end
```

You cannot use the input variables to the constructor to define the default value of the `Angle` property. For example, this definition for the `Angle` property is not valid:

```
properties
      Angle = VectorAngle.Rad2Deg*VectorAngle.getAngle(vx,vy)
end
```

Attempting to create an instance causes an error:

```
a = VectorAngle([1,0],[0,1])
```

```
Error using VectorAngle
Unable to update the class 'VectorAngle' because the new definition contains an
error:
 Undefined function or variable 'vx'.
```

Expressions in Class Methods

Expression in class methods execute like expressions in any function. MATLAB evaluates an expression within the function workspace when the method executes. Therefore, expressions used in class methods are not considered part of the class definition and are not discussed in this section.

How MATLAB Evaluates Expressions

MATLAB evaluates the expressions used in the class definition without any workspace. Therefore, these expressions cannot reference variables of any kind.

MATLAB evaluates expressions in the context of the class file, so these expressions can access any functions, static methods, and constant properties of other classes that are on your path at the time MATLAB initializes the class. Expressions defining property default values can access constant properties defined in their own class.

When MATLAB Evaluates Expressions

MATLAB evaluates the expressions in class definitions only when initializing the class. Initialization occurs before the first use of the class.

After initialization, the values returned by these expressions are part of the class definition and are constant for all instances of the class. Each instance of the class uses the results of the initial evaluation of the expressions without re-evaluation.

If you clear a class, then MATLAB reinitializes the class by reevaluating the expressions that are part of the class definition. (see “Automatic Updates for Modified Classes” on page 5-27)

Expression Evaluation in Handle and Value Classes

The following example shows how value and handle object behave when assigned to properties as default values. Suppose that you have the following classes.

Expressions in Value Classes

The `ClassExp` class has a property that contains a `ContClass` object:

```
classdef ContClass
   properties
      % Assign current date and time
      TimeProp = datestr(now)
   end
end

classdef ClassExp
   properties
      ObjProp = ContClass
   end
end
```

When you first use the `ClassExp` class, MATLAB creates an instance of the `ContClass` class. MATLAB initializes both classes at this time. All instances of `ClassExp` include a copy of this same instance of `ContClass`.

```
a = ClassExp;
a.ObjProp.TimeProp

ans =

08-Oct-2003 17:16:08
```

The `TimeProp` property of the `ContClass` object contains the date and time when MATLAB initialized the class. Creating additional instances of the `ClassExp` class shows that the date string has not changed:

```
b = ClassExp;
b.ObjProp.TimeProp

ans =

08-Oct-2003 17:16:08
```

Because this example uses a value class for the contained object, each instance of the `ClassExp` has its own copy of the object. For example, suppose that you change the value of the `TimeProp` property on the object contained by `ClassExp` object`b`:

```
b.ObjProp.TimeProp = datestr(now)

ans =

08-Oct-2003 17:22:49
```

The copy of the object contained by object `a` is unchanged:

```
a.ObjProp.TimeProp

ans =

08-Oct-2003 17:16:08
```

Expressions in Handle Classes

Now consider the behavior if the contained object is a handle object:

```
classdef ContClass < handle
   properties
      TimeProp = datestr(now)
   end
end
```

Creating two instances of the `ClassExp` class shows that MATLAB created an object when it initialized the `ContClass`. MATLAB used a copy of the object's *handle* for each instance of the `ClassExp` class. Therefore, there is one `ContClass` object and the `ObjProp` property of each `ClassExp` object contains a copy of its handle.

Create an instance of the `ClassExp` class and note the time of creation:

```
a = ClassExp;
a.ObjProp.TimeProp

ans =

08-Oct-2003 17:46:01
```

Create a second instance of the `ClassExp` class. The `ObjProp` contains the handle of the same object:

```
b = ClassExp;
b.ObjProp.TimeProp

ans =

08-Oct-2003 17:46:01
```

Reassign the value of the contained object `TimeProp` property:

```
b.ObjProp.TimeProp = datestr(now);
b.ObjProp.TimeProp

ans =

08-Oct-2003 17:47:34
```

The `ObjProp` property of object `b` contains a handle to the same object as the `ObjProp` property of object `a`. The value of the `TimeProp` property has changed on this object as well:

```
a.ObjProp.TimeProp

ans =

08-Oct-2003 17:47:34
```

See Also

More About

- "Comparison of Handle and Value Classes" on page 7-2

Folders Containing Class Definitions

In this section...

Class Definitions on the Path

To call a class method, the class definition must be on the MATLAB path, as described in the next sections.

Class and Path Folders

There are two types of folders that can contain class definition files.

- Path folders — The folder is on the MATLAB path and the folder name does not begin with an `@` character. Use this type of folder when you want multiple classes and functions in one folder. The entire class definition must be contained in one file.
- Class folders — The folder name begins with an @ character followed by the class name. The folder is not on the MATLAB path, but its parent folder is on the path. Use this type of folder when you want to use multiple files for one class definition.

See the `path` function for information about the MATLAB path.

Using Path Folders

The folders that contain class definition files are on the MATLAB path. Therefore, class definitions placed in path folders behave like any ordinary function with respect to precedence—the first occurrence of a name on the MATLAB path takes precedence over all subsequent occurrences of the same name.

The name of each class definition file must match the name of the class that is specified with the `classdef` keyword. Using a path folder eliminates the need to create a separate class folder for each class. However, the entire class definition, including all methods, must be contained within a single file.

Suppose that you have three classes defined in a single folder:

```
.../path_folder/MyClass1.m
.../path_folder/MyClass2.m
.../path_folder/MyClass3.m
```

To use these classes, add `path_folder` to your MATLAB path:

```
addpath path_folder
```

Using Class Folders

The name of a class folder always begins with the `@` character followed by the class name for the folder name. A class folder must be contained in a path folder, but the class folder is not on the MATLAB path. Place the class definition file inside the class folder, which also can contain separate method files. The class definition file must have the same name as the class folder (without the `@` character).

```
.../parent_folder/@MyClass/MyClass.m
.../parent_folder/@MyClass/myMethod1.m
.../parent_folder/@MyClass/myMethod2.m
```

Define only one class per folder. All files have a `.m` or `.p` extension. For MATLAB versions R2018a and later, standalone methods can be live functions with a `.mlx` extension.

Use a class folder when you want to use more than one file for your class definition. MATLAB treats any function file in the class folder as a method of the class. Function files can be MATLAB code (`.m`), Live Code file format (`.mlx`), MEX functions (platform dependent extensions), and P-code files (`.p`).

MATLAB explicitly identifies any file in a class folder as a method of that class. This enables you to use a more modular approach to authoring methods of your class.

The base name of each file must be a valid MATLAB function name. Valid function names begin with an alphabetic character and can contain letters, numbers, or underscores. For more information, see “Methods in Separate Files” on page 9-8.

Functions in Private Folders Within Class Folders

Private folders contain functions that are accessible only from functions defined in folders immediately above the `private` folder. Any functions defined in a `private` folder inside a class folder can only be called from the methods of the class. The functions have access to the private members of the class but are not themselves methods. They do not require an object to be passed as an input and can only be called using function notation. Use functions in `private` folders when you need helper functions that can be called from multiple methods of your class.

If a class folder contains a `private` folder, only the class defined in that folder can access functions defined in the `private` folder. Subclasses do not have access to superclass private functions. For more information on private folders, see “Private Functions”.

If you want a subclass to have access to the private functions of the superclass, define the functions as protected methods of the superclass. Specify the methods with the `Access` attribute set to `protected`.

Dispatching to Methods in Private Folders

If a class defines functions in a `private` folder that is in a class folder, then MATLAB follows these precedence rules when dispatching to the private functions versus the methods of the `classdef` file:

- Using dot notation (*`obj.methodName`*), a function in a `private` folder takes precedence over a method defined in the `classdef` file.
- Using function notation (*`methodName(obj)`*), a method defined in the `classdef` file takes precedence over the function in the `private` folder.

No Class Definitions in Private Folders

You cannot put class definitions (`classdef` file) in private folders because doing so would not meet the requirements for class or path folders.

Class Precedence and MATLAB Path

When there are multiple class definitions with the same name, the file location on the MATLAB path determines precedence. The class definition in the folder that comes first on the MATLAB path always takes precedence over any classes that are later on the path, whether or not the definitions are contained in a class folder.

A function with the same name as a class in a path folder takes precedence over the class if the function is in a folder that is earlier on the path. However, a class defined in a class folder (@-folder) takes precedence over a function of the same name, even if the function is defined in a folder that is earlier on the path.

For example, consider a path with the following folders and files.

Order in Path	Folder and File	File Defines
1	`fldr1/Foo.m`	Class `Foo`
2	`fldr2/Foo.m`	Function `Foo`
3	`fldr3/@Foo/Foo.m`	Class `Foo`
4	`fldr4/@Foo/bar.m`	Method `bar`
5	`fldr5/Foo.m`	Class `Foo`

MATLAB applies this logic to determine which version of `Foo` to call:

Class `fldr1/Foo.m` takes precedence over the class `fldr3/@Foo` because:

- `fldr1` is before `fldr3` on the path, and `fldr1/Foo.m` is a class.

Class `fldr3/@Foo` takes precedence over function `fldr2/Foo.m` because:

- `fldr3/@Foo` is a class in a class folder.
- `fldr2/Foo.m` is not a class.
- Classes in class folders take precedence over functions.

Function `fldr2/Foo.m` takes precedence over class `fldr5/Foo.m` because:

- `fldr2` comes before class `fldr5` on the path.
- `fldr5/Foo.m` is not in a class folder.
- Classes that are not defined in class folders obey the path order with respect to functions.

Class `fldr3/@Foo` takes precedence over `fldr4/@Foo` because:

- `fldr3` comes before `fldr4` on the path.

If `fldr3/@Foo/Foo.m` contains a MATLAB class created before Version 7.6 (that is, the class does not use the `classdef` keyword), then `fldr4/@Foo/bar.m` becomes a method of the `Foo` class defined in `fldr3/@Foo`.

Previous Behavior of Classes Defined in Class Folders

In MATLAB Versions 5 through 7, class folders do not shadow other class folders having the same name, but residing in later path folders. Instead, the class uses the combination of methods from all class folders having the same name to define the class. This behavior is no longer supported.

For backward compatibility, classes defined in class folders always take precedence over functions and scripts having the same name. This precedence applies to functions and scripts that come before these classes on the path.

Changing Path to Update Class Definition

MATLAB can only recognize one definition of a class as the current definition. Changing your MATLAB path can change the definition file for a class (see `path`). If no instances of the old definition exist (that is, the definition that is no longer first on the path), MATLAB immediately recognizes the new folder as the current definition. If, however, you have an existing instance of the class before changing the path, whether MATLAB uses the definition in the new folder depends on how the new class has been defined. If the new definition is defined in a class folder, MATLAB immediately recognizes the new folder as the current class definition. However, for classes that are defined in path folders (that is, not in class `@` folders), you must clear the class before MATLAB recognizes the new folder as the current class definition.

Class Definitions in Class Folders

Suppose that you define two versions of a class named `Foo` in two folders, `fldA` and `fldB`.

```
fldA/@Foo/Foo.m
fldB/@Foo/Foo.m
```

Add folder `fldA` to the top of the path.

```
addpath fldA
```

Create an instance of class `Foo`. MATLAB uses `fldA/@Foo/Foo.m` as the class definition.

```
a = Foo;
```

Change the current folder to `fldB`.

```
cd fldB
```

The current folder is always first on the path. Therefore, MATLAB finds `fldB/@Foo/Foo.m` as the definition for class `Foo`.

```
b = Foo;
```

MATLAB automatically updates the existing instance, `a`, to use the new class definition in `fldB`.

Class Definitions in Path Folders

Suppose that you define two versions of a class named `Foo` in two folders, `fldA` and `fldB`, but do not use a class folder.

```
fldA/Foo.m
fldB/Foo.m
```

Add folder `fldA` to the top of the path.

```
addpath fldA
```

Create an instance of class `Foo`. MATLAB uses `fldA/Foo.m` as the class definition.

```
a = Foo;
```

Change the current folder to `fldB`.

```
cd fldB
```

The current folder is effectively the top of the path. However, MATLAB does not identify `fldB/Foo.m` as the definition for class `Foo`. MATLAB continues to use the original class definition until you clear the class.

To use the definition of `Foo` in `foldB`, clear `Foo`.

```
clear Foo
```

MATLAB automatically updates the existing objects to conform to the class definition in `fldB`. Usually, clearing instance variables is unnecessary.

See Also

More About

- “Packages Create Namespaces” on page 6-21
- “Automatic Updates for Modified Classes” on page 5-27
- “Live Code File Format (.mlx)”
- “Call MEX Functions”
- “Using MEX Functions for MATLAB Class Methods”
- “Security Considerations to Protect Your Source Code”

Class Precedence

In this section...

Use of Class Precedence

MATLAB uses class precedence to determine which method to call when multiple classes have the same method. You can specify the relative precedence of user-defined classes with the class `InferiorClasses` attribute.

Why Mark Classes as Inferior

When more than one class defines methods with the same name or when classes overload functions, MATLAB determines which method or function to call based on the dominant argument. Here is how MATLAB determines the dominant argument:

1. Determine the dominant argument based on the class of arguments.
2. If there is a dominant argument, call the method of the dominant class.
3. If arguments are of equal precedence, use the leftmost argument as the dominant argument.
4. If the class of the dominant argument does not define a method with the name of the called function, call the first function on the path with that name.

InferiorClasses Attribute

Specify the relative precedence of user-defined classes using the class `InferiorClasses` attribute. To specify classes that are inferior to the class you are defining, assign a cell array of class `meta.class` objects to this attribute.

For example, the following `classdef` declares that `MyClass` is dominant over *ClassName1* and *ClassName2*.

```
classdef (InferiorClasses = {?ClassName1,?ClassName2}) MyClass
   ...
end
```

The `?` operator combined with a class name creates a `meta.class` object. See `metaclass`.

The following MATLAB classes are always inferior to classes defined using the `classdef` syntax and cannot be used in this list.

`double`, `single`, `int64`, `uint64`, `int32`, `uint32`, `int16`, `uint16`, `int8`, `uint8`, `char`, `string`, `logical`, `cell`, `struct`, and `function_handle`.

Dominant Class

MATLAB uses class dominance when evaluating expressions involving objects of more than one class. The dominant class determines:

- Which class method to call when more than one class defines methods with the same names.
- The class of arrays that are formed by combining objects of different classes, assuming MATLAB can convert the inferior objects to the dominant class.

No Attribute Inheritance

Subclasses do not inherit a superclass `InferiorClasses` attribute. Only classes specified in the subclass `InferiorClasses` attribute are inferior to subclass objects.

See Also

More About

- “Class Precedence and MATLAB Path” on page 6-16

Packages Create Namespaces

In this section...

Package Folders

Packages are special folders that can contain class folders, function, and class definition files, and other packages. The names of classes and functions are scoped to the package folder. A package is a namespace within which names must be unique. Function and class names must be unique only within the package. Using a package provides a means to organize classes and functions. Packages also enable you to reuse the names of classes and functions in different packages.

Note Packages are not supported for classes created before MATLAB Version 7.6 (that is, classes that do not use `classdef`).

Package folders always begin with the + character. For example,

```
+mypack
+mypack/pkfcn.m  % a package function
+mypack/@myClass % class folder in a package
```

The parent of the top-level package folder must be on the MATLAB path.

Listing the Contents of a Package

List the contents of a package using the `help` command:

```
help event
```

```
Contents of event:

EventData                     - event.EVENTDATA    Base class for event data
PropertyEvent                 - event.PROPERTYEVENT    Event data for object property events
listener                      - event.LISTENER    Listener object
proplistener                  - event.PROPLISTENER    Listener object for property events
```

You can also use the `what` command:

```
what event
```

```
Classes in directory Y:xxx\matlab\toolbox\matlab\lang\+event

EventData      PropertyEvent  listener       proplistener
```

Internal Packages

MathWorks® reserves the use of packages named `internal` for utility functions used by internal MATLAB code. Functions that belong to an `internal` package are intended for MathWorks use only.

Using functions or classes that belong to an `internal` package is discouraged. These functions and classes are not guaranteed to work in a consistent manner from one release to the next. Any of these functions and classes might be removed from the MATLAB software in any subsequent release without notice and without documentation in the product release notes.

Referencing Package Members Within Packages

All references to packages, functions, and classes in the package must use the package name prefix, unless you import the package. (See “Import Classes” on page 6-25.) For example, call this package function:

```
+mypack/pkfcn.m
```

With this syntax:

```
z = mypack.pkfcn(x,y);
```

Definitions do not use the package prefix. For example, the function definition line of the `pkfcn.m` function would include only the function name:

```
function z = pkfcn(x,y)
```

Define a package class with only the class name:

```
classdef myClass
```

but call it with the package prefix:

```
obj = mypack.myClass(arg1,arg2,...);
```

Calling class methods does not require the package name because you have an object of the class. You can use dot or function notation:

```
obj.myMethod(arg)
myMethod(obj,arg)
```

A static method requires the full class name, which includes the package name:

```
mypack.myClass.stMethod(arg)
```

Referencing Package Members from Outside the Package

Functions, classes, and other packages contained in a package are scoped to that package. To reference any of the package members, prefix the package name to the member name, separated by a dot. For example, the following statement creates an instance of `MyClass`, which is contained in `mypack` package.

```
obj = mypack.MyClass;
```

Accessing Class Members — Various Scenarios

This section shows you how to access various package members from outside a package. Suppose that you have a package `mypack` with the following contents:

```
+mypack
+mypack/myFcn.m
```

```
+mypack/@MyFirstClass
+mypack/@MyFirstClass/myFcn.m
+mypack/@MyFirstClass/otherFcn.m
+mypack/@MyFirstClass/MyFirstClass.m
+mypack/@MySecondClass
+mypack/@MySecondClass/MySecondClass.m
+mypack/+mysubpack
+mypack/+mysubpack/myFcn.m
```

Invoke the `myFcn` function in `mypack`:

```
mypack.myFcn(arg)
```

Create an instance of each class in `mypack`:

```
obj1 = mypack.MyFirstClass;
obj2 = mypack.MySecondClass(arg);
```

Invoke the `myFcn` function that is in the package `mysubpack`:

```
mypack.mysubpack.myFcn(arg1,arg2);
```

If `mypack.MyFirstClass` has a method called `myFcn`, call it like any method call on an object:

```
obj = mypack.MyFirstClass;
myFcn(obj,arg);
```

If `mypack.MyFirstClass` has a property called `MyProp`, assign it using dot notation and the object:

```
obj = mypack.MyFirstClass;
obj.MyProp = x;
```

Packages and the MATLAB Path

You cannot add package folders to the MATLAB path, but you must add the package parent folder to the MATLAB path. Package members are not accessible if the package parent folder is not on the MATLAB path, even if the package folder is the current folder. Making the package folder the current folder is not sufficient to add the package parent folder to the path.

Package members remain scoped to the package. Always refer to the package members using the package name. Alternatively, import the package into the function in which you call the package member, see “Import Classes” on page 6-25.

Package folders do not shadow other package folders that are positioned later on the path, unlike classes, which do shadow other classes. If two or more packages have the same name, MATLAB treats them all as one package. If redundantly named packages in different path folders define the same function name, then MATLAB finds only one of these functions.

Resolving Redundant Names

Suppose a package and a class have the same name. For example:

```
fldr_1/+foo
fldr_2/@foo/foo.m
```

A call to `which foo` returns the path to the executable class constructor:

```
>> which foo
fldr_2/@foo/foo.m
```

A function and a package can have the same name. However, a package name by itself is not an identifier. Therefore, if a redundant name occurs alone, it identifies the function. Executing a package name alone returns an error.

Package Functions vs. Static Methods

In cases where a package and a class have the same name, a package function takes precedence over a static method. For example, path folder `fldrA` contains a package function and path folder `fldrB` contains a class static method:

```
fldrA/+foo/bar.m  % bar is a function in package foo
fldrB/@foo/bar.m  % bar is a static method of class foo
```

A call to `which foo.bar` returns the path to the package function:

```
which foo.bar

fldrA\+foo\bar.m  % package function
```

In cases where the same path folder contains both package and class folders with the same name, the package function takes precedence over the static method.

```
fldr/@foo/bar.m  % bar is a static method of class foo
fldr/+foo/bar.m  % bar is a function in package foo
```

A call to `which foo.bar` returns the path to the package function:

```
which foo.bar

fldr/+foo/bar.m
```

If a path folder `fldr` contains a `classdef` file `foo` that defines a static method `bar` and the same folder contains a package `+foo` that contains a package function `bar`.

```
fldr/foo.m      % bar is a static method of class foo
fldr/+foo/bar.m % bar is a function in package foo
```

A call to `which foo.bar` returns the path to the package function:

```
which foo.bar

fldr/+foo/bar.m
```

See Also

More About

- “Folders Containing Class Definitions” on page 6-14
- “Class Precedence” on page 6-19

Import Classes

In this section...

Syntax for Importing Classes

Import classes into a function to simplify access to class members. For example, suppose that there is a package that contains several classes and you will use only one of these classes or a static method in your function. Use the `import` command to simplify code. Once you have imported the class, you do not need to reference the package name:

```
function myFunc
   import pkg.MyClass
   obj = MyClass(arg,...);                  % call MyClass constructor
   obj.Prop = MyClass.staticMethod(arg,...); % call MyClass static method
end
```

Import all classes in a package using the syntax `pkg.*`:

```
function myFunc
   import pkg.*
   obj1 = MyClass1(arg,...); % call pkg.MyClass1 constructor
   obj2 = MyClass2(arg,...); % call pkg.MyClass2 constructor
   a = pkgFunction();  % call package function named pkgFunction
end
```

Import Static Methods

Use `import` to import a static method so that you can call this method without using the class name. Call `import` with the full class name, including any packages, and the static method name.

```
function myFunc
   import pkg.MyClass.MyStaticMethod
   MyStaticMethod(arg,...); % call static method
end
```

Import Package Functions

Use `import` to import package functions so that you can call these functions without using the package name. Call `import` with the package and function name.

```
function myFunc
   import pkg.pkgFunction
   pkgFunction(arg,...); % call imported package function
end
```

Package Function and Class Method Name Conflict

Avoid importing an entire package using the * wildcard syntax. Doing so imports an unspecified set of names into the local scope. For example, suppose that you have the following folder organization:

```
+pkg/timedata.m           % package function
+pkg/@MyClass/MyClass.m   % class definition file
+pkg/@MyClass/timedata.m  % class method
```

Import the package and call `timedata` on an instance of `MyClass`:

```
import pkg.*
myobj = pkg.MyClass;
timedata(myobj)
```

A call to `timedata` finds the package function, not the class method because MATLAB applies the `import` and finds `pkg.timedata` first. Do not use a package in cases where you have name conflicts and plan to import the package.

Clearing Import List

You cannot clear the import list from a function workspace. To clear the base workspace only, use:

```
clear import
```

See Also

`import`

More About

- “Packages Create Namespaces” on page 6-21

Creating and Managing Class Aliases

A *class alias definition* is a mapping of one or more old class names to a new name. Creating alias definitions enables you to rename classes while maintaining backward compatibility with code and MAT-files that use one or more older class names. In effect, MATLAB recognizes more than one name for the same class. This functionality enables you to update class names when the older names no longer reflect your current design.

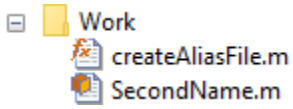
Creating an Alias Definition File

The `matlab.alias.AliasFileManager` class provides an API for creating and maintaining alias definitions. Aliases are not part of class definitions. Instead, alias definition files are stored in `resources` folders that are located in the same folder as the latest class file.

The recommended process for creating and maintaining alias files is to use functions to automate the process. To create a class alias definition, the function must:

1. Create an instance of `matlab.alias.AliasFileManager`.
2. Call the `addAlias` method on the instance with the new class name and the old class names as arguments.
3. Call the `writeAliasFile` method on the instance to write the alias definition file. The method writes the definition file to a folder called `resources`. The method creates the `resources` folder if one does not already exist.

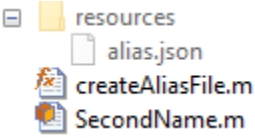
Run the function from the same folder that contains the new class definition. For example, a class named `FirstName` is defined in a folder named `Work`. Update the name of the class from `FirstName` to `SecondName` using a function called `createAliasFile` with this folder structure:




```
function createAliasFile
    fileMgr = matlab.alias.AliasFileManager;
    addAlias(fileMgr,NewName="SecondName",OldNames="FirstName");
    writeAliasFile(fileMgr);
end
```

The `FirstName` class file is not needed to create the alias. In fact, you must remove the original definition from the path so that MATLAB finds the newer alias instead of the older definition.

The final folder structure after running `createAliasFile` looks like this:



MATLAB recognizes both `FirstName` and `SecondName` as the same class as long as `SecondName.m` and the associated alias `resources` folder are in the same folder on the MATLAB path.

See the example “Rename Class” for more details.

Renaming a Class Multiple Times

You can define multiple aliases for the same class. When doing so, you must pass in all of the old aliases to the `addAlias` method. The aliases must be listed in order from newest to oldest. Keeping the original function used to define the alias can help you avoid errors by maintaining a record of the alias definitions. `addAlias` returns an error if all of the previous aliases are not included as part of the `OldNames` input argument.

For an example, see “Rename Class That Has Existing Aliases”.

Renaming a Package

Renaming the entire contents of a package in one step is not supported. To rename a package, you must define aliases for all the classes in the package. For an example, see “Rename Package of Two Classes”.

Note You cannot create aliases for package functions. To rename a package that contains functions, keep the old package in place and redefine the package functions to redirect to the functions in the new package.

Viewing Alias Definitions

There are two ways to view alias definitions:

- Create a `meta.class` instance for the class you want to investigate. The `Aliases` property of `meta.class` returns all of the defined class aliases in a string array, in order from newest to oldest. For more information, see `meta.class`.
- Create a `matlab.alias.AliasFileManager` instance with the `location` input argument, where `location` points to the folder that contains the `resources` folder of the class alias definition file. The `Aliases` property of the `matlab.alias.AliasFileManager` instance returns an array of all alias definitions in that given definition file. For an example of accessing the `Aliases` property, see “Rename Package of Two Classes”.

  Note that reading in an existing alias definition file does not validate that the current class names in that file exist.

Backward and Forward Compatibility of Aliases

To share code that includes classes that have been renamed using aliasing, include one of these two items with your code:

- The alias-creating function in the setup script of your application or toolbox
- The `resources` folder that contains the class alias definition file

The alias definition enables MATLAB to recognize both names as the same class. Including or creating an alias definition in your new code ensures backward compatibility:

- New code can work with code that uses old class names because the old names are recognized as aliases.
- New code can load and work with MAT-files containing serialized objects that were created using the old names.

When serializing objects, using the oldest defined alias for the objects allows the greatest range of compatibility. Objects serialized using the oldest alias can be used by any code back to the original definition and any future code that maintains the full alias definition. Deleting alias definitions or deleting old names from existing definitions is not recommended because backward and forward compatibility can be limited.

See Also

`matlab.alias.AliasFileManager`

Value or Handle Class — Which to Use

Comparison of Handle and Value Classes

In this section...

Basic Difference

A value class constructor returns an object that is associated with the variable to which it is assigned. If you reassign this variable, MATLAB creates an independent copy of the original object. If you pass this variable to a function to modify it, the function must return the modified object as an output argument. For information on value-class behavior, see “Avoid Unnecessary Copies of Data”.

A handle class constructor returns a handle object that is a reference to the object created. You can assign the handle object to multiple variables or pass it to functions without causing MATLAB to make a copy of the original object. A function that modifies a handle object passed as an input argument does not need to return the object.

All handle classes are derived from the abstract `handle` class.

Create a Value Class

By default, MATLAB classes are value classes. The following definition creates a value class named `MyValueClass`:

```
classdef MyValueClass
   ...
end
```

Create a Handle Class

To create a `handle` class, derive the class from the `handle` class.

```
classdef MyHandleClass < handle
   ...
end
```

Behavior of MATLAB Built-In Classes

MATLAB fundamental classes are value classes (numeric, `logical`, `char`, `cell`, `struct`, and function handle). For example, if you create an object of the class `int32` and make a copy of this object, the result is two independent objects. When you change the value of `a`, the value of `b` does not change. This behavior is typical of classes that represent values.

```
a = int32(7);
b = a;
a = a^4;
```

```
b
   7
```

MATLAB graphics objects are implemented as handle objects because they represent visual elements. For example, create a graphics line object and copy its handle to another variable. Both variables refer to the same line object.

```
x = 1:10; y = sin(x);
l1 = line(x,y);
l2 = l1;
```

Set the properties of the line object using either copy of the handle.

```
set(l2,'Color','red')
set(l1,'Color','green')

get(l2,'Color')

ans =

     0     1     0
```

Calling the `delete` function on the `l2` handle destroys the line object. If you attempt to set the `Color` property on the line `l1`, the `set` function returns an error.

```
delete(l2)
set(l1,'Color','blue')

Error using matlab.graphics.primitive.Line/set
Invalid or deleted object.
```

If you delete the object by deleting any one of the existing handles, all copies are now invalid because you deleted the single object to which all handles refer.

Deleting a handle object is not the same as clearing the handle variable. In the graphics object hierarchy, the parent of the object holds a reference to the object. For example, the parent axes hold a reference to the line object referred to by `l1` and `l2`. If you clear both variables from the workspace, the object still exists.

For more information on the behavior of handle objects, see “Handle Object Behavior” on page 1-7.

User-Defined Value Classes

MATLAB associates objects of value classes with the variables to which you assign the object. When you copy a value object to another variable or pass a value object to a function, MATLAB creates an independent copy of the object and all the data contained by the object. The new object is independent of changes to the original object. Value objects behave like MATLAB numeric and `struct` classes. Each property behaves essentially like a MATLAB array.

Value objects are always associated with one workspace or temporary variable. Value objects go out of scope when their variable goes out of scope or is cleared. There are no references to value objects, only copies that are independent objects.

Value Object Behavior

Here is a value class that stores a value in its `Number` property. The default property value is the number `1`.

```
classdef NumValue
   properties
      Number = 1
   end
end
```

Create a `NumValue` object assigned to the variable `a`.

```
a = NumValue
```

```
a =

  NumValue with properties:

    Number: 1
```

Assign the value of `a` to another variable, `b`.

```
b = a
```

```
b =

  NumValue with properties:

    Number: 1
```

The variables `a` and `b` are independent. Changing the value of the `Number` property of `a` does not affect the `Number` property of `b`.

```
a.Number = 7
```

```
a =

  NumValue with properties:

    Number: 7
```

```
b
```

```
b =

  NumValue with properties:

    Number: 1
```

Modifying Value Objects in Functions

When you pass a value object to a function, MATLAB creates a copy of that object in the function workspace. Because copies of value objects are independent, the function does not modify the object in the caller's workspace. Therefore, functions that modify value objects must return the modified object to be reassigned in the caller's workspace.

For more information, see "Object Modification" on page 5-35.

User-Defined Handle Classes

Instances of classes that derive from the `handle` class are references to the underlying object data. When you copy a handle object, MATLAB copies the handle, but does not copy the data stored in the

object properties. The copy refers to the same object as the original handle. If you change a property value on the original object, the copied handle references the same change.

Handle Object Behavior

Here is a handle class that stores a value in its `Number` property. The default property value is the number `1`.

```
classdef NumHandle < handle
   properties
      Number = 1
   end
end
```

Create a `NumHandle` objects assigned to the variable `a`.

```
a = NumHandle
```

```
a =

  NumHandle with properties:

    Number: 1
```

Assign the value of `a` to another variable, `b`.

```
b = a
```

```
b =

  NumHandle with properties:

    Number: 1
```

The variables `a` and `b` refer to the same underlying object. Changing the value of the `Number` property of `a` also changes the `Number` property of `b`. That is, `a` and `b` refer to the same object.

```
a.Number = 7
```

```
a =

  NumHandle with properties:

    Number: 7
```

```
b
```

```
b =

  NumHandle with properties:

    Number: 7
```

Modifying Handle Objects in Functions

When you pass a handle object to a function, MATLAB creates a copy of the handle in the function workspace. Because copies of handles reference the same underlying object, functions that modify the handle object effectively modify the object in the caller's workspace as well. Therefore, it is not

necessary for functions that modify handle objects passed as input arguments to return the modified object to the caller.

For more information, see “Object Modification” on page 5-35.

Deleting Handles

You can destroy handle objects by explicitly calling the handle `delete` method. Deleting the handle of a handle class object makes all handles invalid. For example:

```
a = NumHandle;
b = a;
delete(a)
b.Number
```

```
Invalid or deleted object.
```

Calling `delete` on a handle object invokes the destructor function or functions for that object. See “Handle Class Destructor” on page 7-13 for more information.

Initialize Properties to Contain Handle Objects

For information on the differences between initializing properties to default values in the properties block and initializing properties from within the constructor, see “Initialize Property Values” on page 8-13 and “Initialize Arrays of Handle Objects” on page 10-9.

Determining Equality of Objects

Equality for value objects means that the objects are of the same class and have the same state.

Equality for handle objects means that the handle variables refer to the same object. You also can identify handle variables that refer to different objects of the same class that have the same state.

Equality of Value Objects

To determine if value objects are the same size and their contents are of equal value, use `isequal`. For example, use the previously defined `NumValue` class to create two instances and test for equality:

```
a = NumValue;
b = NumValue;
isequal(a,b)
```

```
ans =

     1
```

`a` and `b` are independent and therefore are not the same object. However each represents the same value.

If you change the value represented by a value object, the objects are no longer equal.

```
a = NumValue;
b = NumValue;
b.Number = 7;
isequal(a,b)
```

```
ans =

     0
```

Value classes do not have a default `eq` method to implement the == operation.

Equality of Handle Objects

Handle objects inherit an `eq` method from the `handle` base class. You can use == and `isequal` to test for two different relationships among handle objects:

- The handles refer to the same object: == and `isequal` return `true`.
- The handles refer to objects of the same class that have the same values, but are not the same objects — only `isequal` returns `true`.

Use the previously defined `NumHandle` class to create an object and copy the handle.

```
a = NumHandle;
b = a;
```

Test for equality using == and `isequal`.

```
a == b
```

```
ans =

     1
```

```
isequal(a,b)
```

```
ans =

     1
```

Create two instances of the `NumHandle` class using the default values.

```
a = NumHandle;
b = NumHandle;
```

Determine if `a` and `b` refer to the same object.

```
a == b
```

```
ans =

     0
```

Determine if `a` and `b` have the same values.

```
isequal(a,b)
```

```
ans =

     1
```

Functionality Supported by Handle Classes

Deriving from the `handle` class enables your class to:

- Inherit several useful methods (“Handle Class Methods” on page 7-11)
- Define events and listeners (“Events and Listeners Syntax” on page 11-18)
- Define dynamic properties (“Dynamic Properties — Adding Properties to an Instance” on page 8-47)
- Implement set and get methods (“Implement Set/Get Interface for Properties” on page 7-22)
- Customize copy behavior (“Implement Copy for Handle Classes” on page 7-30)

See “The Handle Superclass” on page 7-11 for more information on the handle class and its methods.

See Also

Related Examples

- “Which Kind of Class to Use” on page 7-9
- “Implement Copy for Handle Classes” on page 7-30
- “Handle Object Behavior” on page 1-7

Which Kind of Class to Use

In this section...

Examples of Value and Handle Classes

Handle and value classes are useful in different situations. For example, value classes enable you to create array classes that have the same behavior as MATLAB numeric classes.

“Representing Polynomials with Classes” on page 19-2 and “Representing Structured Data with Classes” on page 3-14 provides examples of value classes.

Handle classes enable you to create objects that more than one function or object can share. Handle objects allow more complex interactions among objects because they allow objects to reference each other.

“Implementing Linked Lists with Classes” on page 3-23 and “Developing Classes That Work Together” on page 3-6 provides examples of a handle class.

When to Use Value Classes

Value class objects behave like normal MATLAB variables. A typical use of value classes is to define data structures. For example, suppose that you want to define a class to represent polynomials. This class can define a property to contain a list of coefficients for the polynomial. It can implement methods that enable you to perform various operations on the polynomial object. For example, implement addition and multiplication without converting the object to another class.

A value class is suitable because you can copy a polynomial object and have two objects that are identical representations of the same polynomial. For an example of value classes, see “Subclasses of MATLAB Built-In Types” on page 12-40.

For information on MATLAB pass-by-value semantics, see “Avoid Unnecessary Copies of Data”.

When to Use Handle Classes

Handle objects are useful in specialized circumstances where an object represents a physical object such as a graph or an external device rather than a mathematical object like a number or matrix. Handle objects are derivations of the handle class, which provides functionality such as events and listeners, destructor method, and support for dynamic properties.

Use a handle class when:

- No two instances of a class can have the same state, making it impossible to have exact copies. For example:
  - A copy of a graphics object (such as a line) has a different position in its parents list of children than the object from which it was copied. Therefore, the two objects are not identical.

  - Nodes in lists or trees having specific connectivity to other nodes — no two nodes can have the same connectivity.
- The class represents physical and unique objects like serial ports and printers.
- The class represents visible objects like graphics components.
- The class defines events and notifies listeners when an event occurs (`notify` is a handle class method).
- The class creates listeners by calling the `handle` class `addlistener` method.
- The class subclasses the `dynamicprops` class (a subclass of `handle`) so that instances can define dynamic properties.
- The class subclasses the `matlab.mixin.SetGet` class (a subclass of `handle`) so that it can implement a graphics object style set/get interface to access property values.
- You want to create a singleton class or a class in which you track the number of instances from within the constructor.
- Instances of a class cannot share state, such as nodes in a linked list.

See Also

Related Examples

- “Handle Compatible Classes” on page 12-31

The Handle Superclass

In this section...

Building on the Handle Class

The `handle` class is an abstract class. Therefore, you cannot create objects of this class directly. Use the `handle` class as a superclass to implement subclasses that inherit handle behavior. MATLAB defines several classes that derive from the `handle` class. These classes provide specialized functionality to subclasses.

Specialized Handle Base Classes

To add both handle behavior and specific functionality to your class, derive your class from these `handle` classes:

- `matlab.mixin.SetGet` — Provides `set` and `get` methods to access property values.
- `dynamicprops` — Enables you to define properties that are associated with an object, but not the class in general.
- `matlab.mixin.Copyable` Provides a `copy` method that you can customize for your class.

For information on how to define subclasses, see “Design Subclass Constructors” on page 12-7 .

Handle Class Methods

When you derive a class from the `handle` class, the subclass inherits methods that enable you to work more effectively with handle objects.

List the methods of a class by passing the class name to the `methods` function:

```
methods('handle')
```

```
Methods for class handle:

addlistener  findobj      gt           lt
delete       findprop     isvalid      ne
eq           ge           le           notify
```

Event and Listener Methods

For information on how to use the `notify` and `addlistener` methods, see “Events and Listeners Syntax” on page 11-18.

Relational Methods

```
TF = eq(H1,H2)
TF = ne(H1,H2)
TF = lt(H1,H2)
TF = le(H1,H2)
TF = gt(H1,H2)
TF = ge(H1,H2)
```

The handle class overloads these functions to support equality tests and sorting on handles. For each pair of input arrays, these functions return a logical array of the same size. Each element is an element-wise equality or comparison test result. The input arrays must be the same size or one (or both) can be scalar. The method performs scalar expansion as required. For more information on handle class relational methods, see `relationaloperators`.

Test Handle Validity

Use the `isvalid` `handle` class method to determine if a variable is a valid handle object. For example, in the statement:

```
B = isvalid(H)
```

`B` is a logical array in which each element is `true` if, and only if, the corresponding element of `H` is a valid handle. `B` is always the same size as `H`.

When MATLAB Destroys Objects

MATLAB destroys objects in the workspace of a function when the function:

- Reassigns an object variable to a new value
- Does not use an object variable for the remainder of a function
- Function execution ends

When MATLAB destroys an object, it also destroys values stored in the properties of the object. MATLAB frees computer memory associated with the object for use by MATLAB or the operating system.

You do not need to free memory in handle classes. However, there can be other operations that you want to perform when destroying an object. For example, closing a file or shutting down an external program that the object constructor started. Define a `delete` method in your handle subclass for these purposes.

See “Handle Class Destructor” on page 7-13 for more information.

See Also

Related Examples

- “Comparison of Handle and Value Classes” on page 7-2

Handle Class Destructor

In this section...

Basic Knowledge

Class destructor – a method named `delete` that MATLAB calls implicitly before destroying an object of a handle class. Also, user-defined code can call `delete` explicitly to destroy an object.

Nondestructor – a method named `delete` that does not meet the syntax requirements of a valid destructor. Therefore, MATLAB does not call this method implicitly when destroying handle objects. A method named `delete` in a value class is not a destructor. A method named `delete` in a value class that sets the `HandleCompatible` attribute to `true` is not a destructor.

“Object Lifecycle” on page 7-16

“Method Attributes” on page 9-4

Syntax of Handle Class Destructor Method

MATLAB calls the destructor of a handle class when destroying objects of the class. MATLAB recognizes a method named `delete` as the class destructor only if you define `delete` as an ordinary method with the appropriate syntax.

To be a valid class destructor, the `delete` method:

- Must define one, scalar input argument, which is an object of the class.
- Must not define output arguments
- Cannot be `Sealed`, `Static`, or `Abstract`
- Cannot use `arguments` blocks for input argument validation.

In addition, the `delete` method should *not*:

- Throw errors, even if the object is invalid.
- Create new handles to the object being destroyed

- Call methods or access properties of subclasses

MATLAB does not call a noncompliant `delete` method when destroying objects of the class. A noncompliant `delete` method can prevent the destruction of the object by shadowing the `handle` class delete method.

A `delete` method defined by a value class that is handle compatible is not a destructor, even if the `delete` method is inherited by a handle subclass. For information on handle compatible classes, see “Handle Compatible Classes” on page 12-31.

Declare `delete` as an ordinary method:

```
methods
   function delete(obj)
      % obj is always scalar
   ...
   end
end
```

delete Called Element-Wise on Array

MATLAB calls the `delete` method separately for each element in an array. Therefore, a `delete` method is passed only one scalar argument with each invocation.

Calling `delete` on a deleted handle should not error and can take no action. This design enables `delete` to work on object arrays containing a mix of valid and invalid objects.

Handle Object During delete Method Execution

Calling the `delete` method on an object always results in the destruction of the object. The object is destroyed when the call to `delete` is made explicitly in MATLAB code or when called by MATLAB because an object is no longer reachable from any workspace. Once called, a `delete` method cannot abort or prevent object destruction.

A `delete` method can access properties of the object being deleted. MATLAB does not destroy these properties until after the `delete` methods for the class of the object and all superclasses finish executing.

If a `delete` method creates new variables that contain a handle to the object being deleted, those handles are invalid. After the `delete` method finishes execution, handles to the deleted object in any variables in any workspace are invalid.

The `isvalid` method returns `false` for the handle object within the `delete` method because object destruction begins when the method is called.

MATLAB calls `delete` methods in the inverse of the construction order. That is, MATLAB invokes subclass `delete` methods before superclass `delete` methods.

If a superclass expects a property to be managed by subclasses, then the superclass should not access that property in its `delete` method. For example, if a subclass uses an inherited abstract property to store an object handle, then the subclass should destroy this object in its `delete` method, but the superclass should not access that property in its `delete` method.

Support Destruction of Partially Constructed Objects

Errors that occur while constructing an object can result in a call to `delete` before the object is fully created. Therefore, class `delete` methods must be able to work with partially constructed objects.

For example, the `PartialObject` class `delete` method determines if the `Data` property is empty before accessing the data this property contains. If an error occurs while assigning the constructor argument to the `Name` property, MATLAB passes the partially constructed object to delete.

```
classdef PartialObject < handle
   properties
      % Restrict the Name property
      % to a cell array
      Name cell
      Data
   end
   methods
      function h = PartialObject(name)
         if nargin > 0
            h.Name = name;
            h.Data.a = rand(10,1);
         end
      end
      function delete(h)
         % Protect against accessing properties
         % of partially constructed objects
         if ~isempty(h.Data)
            t = h.Data.a;
            disp(t)
         else
            disp('Data is empty')
         end
      end
   end
end
```

An error occurs if you call the constructor with a `char` vector, instead of the required cell array:

```
obj = PartialObject('Test')
```

MATLAB passes the partially constructed object to the `delete` method. The constructor did not set the value of the `Data` property because the error occurred when setting the `Name` property.

```
Data is empty
Error setting 'Name' property of 'PartialObject' class:
...
```

When to Define a Destructor Method

Use a `delete` method to perform cleanup operations before MATLAB destroys the object. MATLAB calls the `delete` method reliably, even if execution is interrupted with Ctrl-c or an error.

If an error occurs during the construction of a handle class, MATLAB calls the class destructor on the object along with the destructors for any objects contained in properties and any initialized base classes.

For example, suppose that a method opens a file for writing and you want to close the file in your `delete` method. The `delete` method can call `fclose` on a file identifier that the object stores in its `FileID` property:

```
function delete(obj)
   fclose(obj.FileID);
end
```

Destructors in Class Hierarchies

If you create a hierarchy of classes, each class can define its own `delete` method. When destroying an object, MATLAB calls the `delete` method of each class in the hierarchy. Defining a `delete` method in a `handle` subclass does not override the `handle` class `delete` method. Subclass `delete` methods augment the superclass `delete` methods.

Inheriting a Sealed Delete Method

Classes cannot define a valid destructor that is `Sealed`. MATLAB returns an error when you attempt to instantiate a class that defines a `Sealed` `delete` method.

Normally, declaring a method as `Sealed` prevents subclasses from overriding that method. However, a `Sealed` method named `delete` that is not a valid destructor does not prevent a subclass from defining its own destructor.

For example, if a superclass defines a method named `delete` that is not a valid destructor, but is `Sealed`, then subclasses:

- Can define valid destructors (which are always named `delete`).
- Cannot define methods named `delete` that are not valid destructors.

Destructors in Heterogeneous Hierarchies

Heterogeneous class hierarchies require that all methods to which heterogeneous arrays are passed must be sealed. However, the rule does not apply to class destructor methods. Because destructor methods cannot be sealed, you can define a valid destructor in a heterogeneous hierarchy that is not sealed, but does function as a destructor.

For information on heterogeneous hierarchies, see “Designing Heterogeneous Class Hierarchies” on page 10-22

Object Lifecycle

MATLAB invokes the `delete` method when the lifecycle of an object ends. The lifecycle of an object ends when the object is:

- No longer referenced anywhere
- Explicitly deleted by calling `delete` on the handle

Inside a Function

The lifecycle of an object referenced by a local variable or input argument exists from the time the variable is assigned until the time it is reassigned, cleared, or no longer referenced within that function or in any handle array.

A variable goes out of scope when you explicitly clear it or when its function ends. When a variable goes out of scope and its value belongs to a handle class that defines a `delete` method, MATLAB calls that method. MATLAB defines no ordering among variables in a function. Do not assume that MATLAB destroys one value before another value when the same function contains multiple values.

Sequence During Handle Object Destruction

MATLAB invokes the `delete` methods in the following sequence when destroying an object:

1. The `delete` method for the class of the object
2. The `delete` method of each superclass class, starting with the immediate superclasses and working up the hierarchy to the most general superclasses

MATLAB invokes the `delete` methods of superclasses at the same level in the hierarchy in the order specified in the class definition. For example, the following class definition specifies `supclass1` before `supclass2`. MATLAB calls the `delete` method of `supclass1` before the `delete` method of `supclass2`.

```
classdef myClass < supclass1 & supclass2
```

After calling each `delete` method, MATLAB destroys the property values belonging exclusively to the class whose method was called. The destruction of property values that contain other handle objects can cause a call the `delete` methods for those objects when there are no other references to those objects.

Superclass `delete` methods cannot call methods or access properties belonging to a subclass.

Destruction of Objects with Cyclic References

Consider a set of objects that reference other objects of the set such that the references form a cyclic graph. In this case, MATLAB:

- Destroys the objects if they are referenced only within the cycle
- Does not destroy the objects as long as there is an external reference to any of the objects from a MATLAB variable outside the cycle

MATLAB destroys the objects in the reverse of the order of construction. for more information, see “Handle Object During delete Method Execution” on page 7-14.

Restrict Access to Object Delete Method

Destroy handle objects by explicitly calling `delete` on the object:

```
delete(obj)
```

A class can prevent explicit destruction of an object by setting its `delete` method `Access` attribute to `private`. However, a method of the class can call the `private` `delete` method.

If the class `delete` method `Access` attribute is `protected`, only methods of the class and of subclasses can explicitly delete objects of that class.

However, when an object lifecycle ends, MATLAB calls the object’s `delete` method when destroying the object regardless of the method’s `Access` attribute.

Inherited Private Delete Methods

Class destructor behavior differs from the normal behavior of an overridden method. MATLAB executes each `delete` method of each superclass upon destruction, even if that `delete` method is not `public`.

When you explicitly call an object's `delete` method, MATLAB checks the `delete` method `Access` attribute in the class defining the object, but not in the superclasses of the object. A superclass with a `private` `delete` method cannot prevent the destruction of subclass objects.

Declaring a private delete method makes most sense for sealed classes. In the case where classes are not sealed, subclasses can define their own delete methods with public access. MATLAB calls a private superclass `delete` method as a result of an explicit call to a public subclass `delete` method.

Nondestructor Delete Methods

A class can implement a method named `delete` that is not a valid class destructor. MATLAB does not call this method implicitly when destroying an object. In this case, `delete` behaves like an ordinary method.

For example, if the superclass implements a `Sealed` method named `delete` that is not a valid destructor, then MATLAB does not allow subclasses to override this method.

A `delete` method defined by a value class cannot be a class destructor.

External References to MATLAB Objects

MATLAB does not manage object lifecycles that involve external languages that perform their own object lifecycle management (aka, garbage collection). MATLAB cannot detect when it is safe to destroy objects used in cyclic references because the external environment does not notify MATLAB when external reference have been destroyed.

If you cannot avoid external references to MATLAB objects, explicitly break the cyclic reference by destroying the objects in MATLAB.

The following section describes how to manage this situation when using Java objects that reference MATLAB objects.

Java References Can Prevent Destructor Execution

Java does not support the object destructors that MATLAB objects use. Therefore, it is important to manage the lifecycle of all objects used in applications that include both Java and MATLAB objects.

Java objects that hold references to MATLAB objects can prevent deletion of the MATLAB objects. In these cases, MATLAB does not call the handle object `delete` method even when there is no handle variable referring to that object. To ensure your `delete` method executes, call `delete` on the object explicitly before the handle variable goes out of scope.

Problems can occur when you define callbacks for Java objects that reference MATLAB objects.

For example, the `CallbackWithJava` class creates a Java `com.mathworks.jmi.Callback` object and assigns a class method as the callback function. The result is a Java object that has a reference to a handle object via the function-handle callback.

```
classdef CallbackWithJava < handle
   methods
      function obj = CallbackWithJava
         jo = com.mathworks.jmi.Callback;
         set(jo,'DelayedCallback',@obj.cbFunc); % Assign method as callback
         jo.postCallback
      end
      function cbFunc(obj,varargin)
         c = class(obj);
         disp(['Java object callback on class ',c])
      end
      function delete(obj)
         c = class(obj);
         disp(['ML object destructor called for class ',c])
      end
   end
end
```

Suppose that you create a `CallbackWithJava` object from within a function:

```
function testDestructor
   cwj = CallbackWithJava
   ...
end
```

Creating an instance of the `CallbackWithJava` class creates the `com.mathworks.jmi.Callback` object and executes the callback function:

```
testDestructor
```

```
cwj =

  CallbackWithJava with no properties.

Java object callback on class CallbackWithJava
```

The handle variable, `cwj`, exists only in the function workspace. However, MATLAB does not call the class `delete` method when the function ends. The `com.mathworks.jmi.Callback` object still exists and holds a reference to the object of the `CallbackWithJava` class, which prevents destruction of the MATLAB object.

```
clear classes
```

```
Warning: Objects of 'CallbackWithJava' class exist.  Cannot clear this class or
any of its superclasses.
```

To avoid causing inaccessible objects, call `delete` explicitly before losing the handle to the MATLAB object.

```
function testDestructor
   cwj = CallbackWithJava
   ...
   delete(cwj)
end
```

Manage Object Lifecycle in Applications

MATLAB applications that use Java or other external-language objects should manage the lifecycle of the objects involved. A typical user interface application references Java objects from MATLAB objects and creates callbacks on Java objects that reference MATLAB objects.

You can break these cyclic references in various ways:

- Explicitly call `delete` on the MATLAB objects when they are no longer needed
- Unregister the Java object callbacks that reference MATLAB objects
- Use intermediate handle objects that reference both the Java callbacks and the MATLAB objects.

See Also

More About

- “Handle Object Behavior” on page 1-7

Find Handle Objects and Properties

In this section...

Find Handle Objects

The `findobj` method enables you to locate handle objects that meet certain conditions.

```
function HM = findobj(H,<conditions>)
```

The `findobj` method returns an array of handles matching the conditions specified. You can use regular expressions with `findobj`. For more information, see `regexp`.

Find Handle Object Properties

The `findprop` method returns the `meta.property` object for the specified object and property.

```
function mp = findprop(h,'PropertyName')
```

The property can also be a dynamic property created by the `addprop` method of the `dynamicprops` class.

Use the returned `meta.property` object to obtain information about the property, such as the settings of any of its attributes. For example, the following statements determine that the setting of the `AccountStatus` property `Dependent` attribute is `false`.

```
ba = BankAccount(007,50,'open');
mp = findprop(ba,'AccountStatus');
mp.Dependent

ans =
     0
```

See Also

`handle`

Related Examples

- “Class Metadata” on page 16-2

Implement Set/Get Interface for Properties

In this section...

The Standard Set/Get Interface

Some MATLAB objects, such as graphics objects, implement an interface based on `set` and `get` functions. These functions enable access to multiple properties on arrays of objects in a single function call.

You can add `set` and `get` functionality to your class by deriving from one of these classes:

- `matlab.mixin.SetGet` — use when you want support for case-insensitive, partial property name matching. Deriving from `matlab.mixin.SetGet` does not affect the exact property name required by the use of dot notation reference to properties.
- `matlab.mixin.SetGetExactNames` — use when you want to support only case-sensitive full property name matching.

Note The `set` and `get` methods referred to in this section are different from property set access and property get access methods. See “Property Get and Set Methods” on page 8-38 for information on property access methods.

Subclass Syntax

Use the abstract class `matlab.mixin.SetGet` or `matlab.mixin.SetGetExactNames` as a superclass:

```
classdef MyClass < matlab.mixin.SetGet
   ...
end
```

Because `matlab.mixin.SetGet` and `matlab.mixin.SetGetExactNames` derive from the `handle` class, your subclass is also a `handle` class.

Get Method Syntax

The `get` method returns the value of an object property using the object handle and the property name. For example, assume `H` is the handle to an object:

```
v = get(H,'PropertyName');
```

If you specify an array of handles with a single property name, `get` returns the property value for each object as a cell array of values:

```
CV = get(H,'PropertyName');
```

The `CV` array is always a column regardless of the shape of `H`.

If you specify a cell array of `char` vector property names and an array of handles, `get` returns a cell array of property values. Each row in the cell corresponds to an object in the handle array. Each column in the cell corresponds to a property name.

```
props = {'PropertyName1','PropertyName2'};
CV = get(H,props);
```

`get` returns an m-by-n cell array, where `m = length(H)` and `n = length(props)`.

If you specify a handle array, but no property names, `get` returns an array of type `struct` in which each structure in the array corresponds to an object in `H`. Each field in each structure corresponds to a property defined by the class of `H`. The value of each field is the value of the corresponding property.

```
SV = get(H);
```

If you do not assign an output variable, then `H` must be scalar.

For an example, see “Using get with Arrays of Handles” on page 7-25.

Set Method Syntax

The `set` method assigns the specified value to the specified property for the object with handle `H`. If `H` is an array of handles, MATLAB assigns the value to the property for each object in the array `H`.

```
set(H,'PropertyName',PropertyValue)
```

You can pass a cell array of property names and a cell array of property values to `set`:

```
props = {'PropertyName1','PropertyName2'};
vals = {Property1Value,Property2Value};
set(H,props,vals)
```

If `length(H)` is greater than one, then the property value cell array (`vals`) can have values for each property in each object. For example, suppose `length(H)` is 2 (two object handles). You want to assign two property values on each object:

```
props = {'PropertyName1','PropertyName2'};
vals = {Property11Value,Property12Value;Property21Value,Property22Value};
set(H,props,vals))
```

The preceding statement is equivalent to the follow two statements:

```
set(H(1),'PropertyName1',Property11Value,'PropertyName2',Property12Value)
set(H(2),'PropertyName1',Property21Value,'PropertyName2',Property22Value)
```

If you specify a scalar handle, but no property names, `set` returns a `struct` with one field for each property in the class of `H`. Each field contains an empty cell array.

```
SV = set(h);
```

Tip You can use any combination of property name/property value cell arrays, structure arrays (with the field names as property names and field values as property values), and cell arrays in one call to `set`.

Class Derived from matlab.mixin.SetGet

This sample class defines a set/get interface and illustrates the behavior of the inherited methods:

```
classdef LineType < matlab.mixin.SetGet
   properties
      Style = '-'
      Marker = 'o'
   end
   properties (SetAccess = protected)
      Units = 'points'
   end
   methods
      function obj = LineType(s,m)
         if nargin > 0
            obj.Style = s;
            obj.Marker = m;
         end
      end
      function set.Style(obj,val)
         if ~(strcmpi(val,'-') ||...
               strcmpi(val,'--') ||...
               strcmpi(val,'..'))
            error('Invalid line style ')
         end
         obj.Style = val;
      end
      function set.Marker(obj,val)
         if ~isstrprop(val,'graphic')
            error('Marker must be a visible character')
         end
         obj.Marker = val;
      end
   end
end
```

Create an instance of the class and save its handle:

```
h = LineType('--','*');
```

Query the value of any object property using the inherited `get` method:

```
get(h,'Marker')
```

```
ans =

'*'
```

Set the value of any property using the inherited `set` method:

```
set(h,'Marker','Q')
```

Property Access Methods Called with set and get

MATLAB calls property access methods (`set.Style` or `set.Marker` in the `LineType` class) when you use the `set` and `get` methods.

```
set(h,'Style','-.-')
```

```
Error using LineType/set.Style (line 20)
Invalid line style
```

For more information on property access methods, see “Property Get and Set Methods” on page 8-38

List All Properties

Return a `struct` containing object properties and their current values using `get`:

```
h = LineType('--','*');
SV = get(h)
```

```
SV =

  struct with fields:

     Style: '--'
    Marker: '*'
     Units: 'points'
```

Return a `struct` containing the properties that have `public` `SetAccess` using `set`:

```
S = set(h)
```

```
S =

  struct with fields:

     Style: {}
    Marker: {}
```

The `LineType` class defines the `Units` property with `SetAccess = protected`. Therefore, `S = set(h)` does not create a field for `Units` in `S`.

`set` cannot return possible values for properties that have nonpublic set access.

Using get with Arrays of Handles

Suppose that you create an array of `LineType` objects:

```
H = [LineType('..','z'),LineType('--','q')]
```

```
H =

  1x2 LineType with properties:

    Style
    Marker
    Units
```

When `H` is an array of handles, `get` returns a (`length(H)`-by-1) cell array of property values:

```
CV = get(H,'Style')
```

```
CV =

  2×1 cell array
```

```
    {'..'}
    {'--'}
```

When H is an array of handles and you do not specify a property name, `get` returns a `struct` array containing fields with names corresponding to property-names. Assign the output of `get` to a variable when H is not scalar.

```
SV = get(H)

SV =

2x1 struct array with fields:
    Style
    Marker
    Units
```

Get the value of the `Marker` property from the second array element in the SV array of structures:

```
SV(2).Marker

ans =

'q'
```

Arrays of Handles, Names, and Values

You can pass an array of handles, a cell array of property names, and a cell array of property values to `set`. The property value cell array must have one row of property values for each object in H. Each row must have a value for each property in the property name array:

```
H = [LineType('..','z'),LineType('--','q')];
set(H,{'Style','Marker'},{'..','o';'--','x'})
```

The result of this call to `set` is:

```
H(1)

ans =

  LineType with properties:

     Style: '..'
    Marker: 'o'
     Units: 'points

H(2)

ans =

  LineType with properties:

     Style: '--'
    Marker: 'x'
     Units: 'points'
```

Customize the Property List

Customize the way property lists display by redefining the following methods in your subclass:

- `setdisp` — When you call `set` with no output argument and a single scalar handle input, `set` calls `setdisp` to determine how to display the property list.
- `getdisp` — When you call `get` with no output argument and a single scalar handle input, `get` calls `getdisp` to determine how to display the property list.

Set Priority for Matching Partial Property Names

Classes that derive from `matlab.mixin.SetGet` can use the `PartialMatchPriority` property attribute to specify a relative priority for partial name matching. MATLAB applies this attribute when resolving incomplete and case-insensitive text strings that match more than one property name.

The inherited `set` and `get` methods can resolve inexact property names when there are no ambiguities resulting from inexact name strings. When a partial property name is ambiguous because the name matches more than one property, the `PartialMatchPriority` attribute value can determine which property MATLAB matches.

The default priority is equivalent to `PartialMatchPriority = 1`. To reduce the relative priority of a property, set `PartialMatchPriority` to a positive integer value of 2 or greater. The priority of a property decreases as the value of `PartialMatchPriority` increases.

For example, in this class the `Verbosity` property has a higher priority for name matching than the `Version` property.

```
classdef MyClass < matlab.mixin.SetGet
    properties
        Verbosity
    end
    properties (PartialMatchPriority = 2)
        Version
    end
end
```

Calling the `set` method with the potentially ambiguous inexact name `Ver` sets the `Verbosity` property because of its higher relative priority. Without setting the `PartialMatchPriority` attribute, the ambiguous name would cause an error.

```
a = MyClass;
set(a,"Ver",10)
disp(a)
```

```
 MyClass with properties:

    Verbosity: 10
      Version: []
```

The same name selection applies to the `get` method.

```
v = get(a,"Ver")
```

```
v =

    10
```

Case and Name Matching

A full name match with nonmatching case takes precedence over a partial match with a higher priority property. For example, this class defines the `BaseLine` property with a priority of 1 (the default) and a `Base` property with a priority of 2 (lower than 1).

```
classdef MyClass < matlab.mixin.SetGet
    properties
        BaseLine
    end
    properties (PartialMatchPriority = 2)
        Base
    end
end
```

Calling the `set` method with the string `base` sets the `Base` property. `BaseLine` has a higher priority, but the full name match with incorrect case takes precedence.

```
a = MyClass;
set(a,"base",-2)
disp(a)
```

```
 MyClass with properties:

    BaseLine: []
        Base: -2
```

Reduce Incompatibilities When Adding New Properties

You can use the `PartialMatchPriority` attribute to avoid introducing code incompatibilities when adding a new property. For example, this class enables the `set` and `get` methods to refer to the `Distance` property with the string `Dis` because the `DiscreteSamples` property has a lower priority.

```
classdef Planet < matlab.mixin.SetGet
% Version 1.0
    properties
        Distance
    end
    properties(PartialMatchPriority = 2)
        DiscreteSamples
    end
end
```

Version 2.0 of the class introduces a property named `Discontinuities`. To prevent the possibility of causing an ambiguous partial property name in existing code, use `PartialMatchPriority` to set the priority of `Discontinuities` lower than that of previously existing properties.

```
classdef Planet < matlab.mixin.SetGet
% Version 2.0
    properties
        Diameter;
        NumMoons = 0
        ApparentMagnitude;
        DistanceFromSun;
    end
    properties(PartialMatchPriority = 2)
        DiscreteSamples;
```

```
    end
    properties(PartialMatchPriority = 3)
        Discontinuities = false;
    end
end
```

For version 1.0 of the `Planet` class, this call to the `set` method was not ambiguous.

```
p = Planet;
set(p,"Disc",true)
```

However, with the introduction of the `Discontinuities` property, the string `Disc` becomes ambiguous. By giving the `Discontinuities` property a lower priority, the string `Disc` continues to match the `DiscreteSamples` property.

Note When writing reusable code, using complete, case-sensitive property names avoids ambiguities, prevents incompatibilities with subsequent software releases, and produces more readable code.

See Also

`set` | `get` | `matlab.mixin.SetGet` | `matlab.mixin.SetGetExactNames`

More About

- “Ways to Use Properties” on page 8-2

Implement Copy for Handle Classes

In this section...

Copy Method for Handle Classes

Copying a handle variable results in another handle variable that refers to the same object. You can add copy functionality to your handle class by subclassing `matlab.mixin.Copyable`. The inherited `copy` method enables you to make shallow copies of objects of the class. The `CopyObj` class shows the behavior of copy operations.

```
classdef CopyObj < matlab.mixin.Copyable
   properties
      Prop
   end
end
```

Create an object of the `CopyObj` class and assign the handle of a `line` object to the property `Prop`.

```
a = CopyObj;
a.Prop = line;
```

Copy the object.

```
b = copy(a);
```

Confirm that the handle variables `a` and `b` refer to different objects.

```
a == b
```

```
ans =

  logical

   0
```

However, the `line` object referred to by `a.Prop` has not been copied. The handle contained in `a.Prop` refers to the same object as the handle contained in `b.Prop`.

```
a.Prop == b.Prop
```

```
ans =

  logical

   1
```

For more detailed information on the behavior of the copy operation, see `copy`.

Customize Copy Operation

Customize handle object copy behavior by deriving your class from `matlab.mixin.Copyable`. The `matlab.mixin.Copyable` class is an abstract base class that derives from the handle class. `matlab.mixin.Copyable` provides a template for customizing object copy operations by defining:

- `copy` — Sealed method that defines the interface for copying objects
- `copyElement` — Protected method that subclasses can override to customize object copy operations for the subclass

The `matlab.mixin.Copyable` `copy` method, calls the `copyElement` method. Your subclass customizes the copy operation by defining its own version of `copyElement`.

The default implementation of `copyElement` makes shallow copies of all the nondependent properties. `copyElement` copies each property value and assigns it to the new (copied) property. If a property value is a handle object, `copyElement` copies the handle, but not the underlying data.

To implement different copy behavior for different properties, override `copyElement`. For example, the `copyElement` method of the `SpecializedCopy` class:

- Creates a new class object
- Copies the value of `Prop1` to the new object
- Reinitializes the default value of `Prop2` by adding a timestamp when the copy is made

```
classdef SpecializedCopy < matlab.mixin.Copyable
   properties
      Prop1
      Prop2 = datestr(now)
   end
   methods(Access = protected)
      function cp = copyElement(obj)
         cp = SpecializedCopy;
         cp.Prop1 = obj.Prop1;
         cp.Prop2 = datestr(now);
      end
   end
end
```

Create an object of the class and assign a value to `Prop1`:

```
a  = SpecializedCopy;
a.Prop1 = 7
```

```
a =

  SpecializedCopy with properties:

    Prop1: 7
    Prop2: '17-Feb-2015 17:51:23'
```

Use the inherited `copy` method to create a copy of `a`:

```
b = copy(a)
```

```
b =
```

```
  SpecializedCopy with properties:

    Prop1: 7
    Prop2: '17-Feb-2015 17:51:58'
```

The copy (object `b`) has the same value for `Prop1`, but the subclass `copyElement` method assigned a new value to `Prop2`. Notice the different timestamp.

Copy Properties That Contain Handles

Copying an object also copies the values of object properties. Object properties can contain other objects, including handle objects. If you simply copy the value of a property that contains a handle object, you are actually copying the handle, not the object itself. Therefore, your copy references the same object as the original object. Classes that derive from the `matlab.mixin.Copyable` class can customize the way the copy method copies objects of the class.

Class to Support Handle Copying

Suppose that you define a class that stores a handle in an object property. You want to be able to copy objects of the class and want each copy of an object to refer to a new handle object. Customize the class copy behavior using these steps:

- Create a subclass of `matlab.mixin.Copyable`.
- Override `copyElement` to control how the property containing the handle is copied.
- Because the property value is a handle, create a new default object of the same class.
- Copy property values from the original handle object to the new handle object.

The “HandleCopy” on page 7-32 class customizes copy operations for the property that contains a handle object. The “ColorProp” on page 7-33 class defines the handle object to assign to `Prop2`:

Create an object and assign property values:

```
a = HandleCopy;
a.Prop1 = 7;
a.Prop2 = ColorProp;
```

Make a copy of the object using the `copy` method inherited from `matlab.mixin.Copyable`:

```
b = copy(a);
```

Demonstrate that the handle objects contained by objects `a` and `b` are independent. Changing the value on object `a` does not affect object `b`:

```
a.Prop2.Color = 'red';
b.Prop2.Color

ans =

blue
```

HandleCopy

The `HandleCopy` class customizes the copy operation for objects of this class.

```
classdef HandleCopy < matlab.mixin.Copyable
   properties
```

```
      Prop1 % Shallow copy
      Prop2 % Handle copy
   end
   methods (Access = protected)
      function cp = copyElement(obj)
         % Shallow copy object
         cp = copyElement@matlab.mixin.Copyable(obj);
         % Get handle from Prop2
         hobj = obj.Prop2;
         % Create default object
         new_hobj = eval(class(hobj));
         % Add public property values from orig object
         HandleCopy.propValues(new_hobj,hobj);
         % Assign the new object to property
         cp.Prop2  = new_hobj;
      end
   end
   methods (Static)
      function propValues(newObj,orgObj)
         pl = properties(orgObj);
         for k = 1:length(pl)
            if isprop(newObj,pl{k})
               newObj.(pl{k}) = orgObj.(pl{k});
            end
         end
      end
   end
end
```

ColorProp

The `ColorProp` class defines a color by assigning an RGB value to its `Color` property.

```
classdef ColorProp < handle
   properties
      Color = 'blue';
   end
end
```

Exclude Properties from Copy

Use the `NonCopyable` property attribute to indicate that you do not want a copy operation to copy a particular property value. By default, `NonCopyable` is `false`, indicating that the property value is copyable. You can set `NonCopyable` to `true` only on properties of handle classes.

For classes that derive from `matlab.mixin.Copyable`, the default implementation of `copyElement` honors the `NonCopyable` attribute. Therefore, if a property has its `NonCopyable` attribute set to `true`, then `copyElement` does not copy the value of that property. If you override `copyElement` in your subclass, you can choose how to use the `NonCopyable` attribute.

Set the Attribute to Not Copy

Set `NonCopyable` to `true` in a property block:

```
properties (NonCopyable)
   Prop1
end
```

Default Values

If a property that is not copyable has a default value assigned in the class definition, the copy operation assigns the default value to the property. For example, the `CopiedClass` assigns a default value to `Prop2`.

```
classdef CopiedClass < matlab.mixin.Copyable
   properties (NonCopyable)
      Prop1
      Prop2 = datestr(now) % Assign current time
   end
end
```

Create an object to copy and assign a value to `Prop1`:

```
a = CopiedClass;
a.Prop1 = 7
```

```
a =

  CopiedClass with properties:

    Prop1: 7
    Prop2: '17-Feb-2015 15:19:34'
```

Copy `a` to `b` using the `copy` method inherited from `matlab.mixin.Copyable`:

```
b = copy(a)
```

```
b =

  CopiedClass with properties:

    Prop1: []
    Prop2: '17-Feb-2015 15:19:34'
```

In the copy `b`, the value of `Prop1` is not copied. The value of `Prop2` is set to its default value, which MATLAB determined when first loading the class. The timestamp does not change.

Objects with Dynamic Properties

Subclasses of the `dynamicprops` class allow you to add properties to an object of the class. When a class derived from `dynamicprops` is also a subclass of `matlab.mixin.Copyable`, the default implementation of `copyElement` does not copy dynamic properties. The default value of `NonCopyable` is `true` for dynamic properties.

The default implementation of `copyElement` honors the value of a dynamic property `NonCopyable` attribute. If you want to allow copying of a dynamic property, set its `NonCopyable` attribute to `false`. Copying a dynamic property copies the property value and the values of the property attributes.

For example, this copy operation copies the dynamic property, `DynoProp`, because its `NonCopyable` attribute is set to `false`. The object `obj` must be an instance of a class that derives from both `dynamicprops` and `matlab.mixin.Copyable`:

```
obj = MyDynamicClass;
p = addprop(obj,'DynoProp');
```

```
p.NonCopyable = false;
obj2 = copy(obj);
```

See Also

`matlab.mixin.Copyable`

Related Examples

- “Dynamic Properties — Adding Properties to an Instance” on page 8-47

Properties — Storing Class Data

Ways to Use Properties

In this section...

What Are Properties

Properties encapsulate the data that belongs to instances of classes. Data contained in properties can be public, protected, or private. This data can be a fixed set of constant values, or it can depend on other values and calculated only when queried. You control these aspects of property behaviors by setting property attributes and by defining property-specific access methods.

Flexibility of Object Properties

In some ways, properties are like fields of a `struct` object. However, storing data in an object property provides more flexibility. Properties can:

- Define a constant value that you cannot change outside the class definition. See “Define Class Properties with Constant Values” on page 15-2.
- Calculate its own value based on the current value of other data. See “Features of Dependent Properties” on page 8-3.
- Execute a function to determine if an attempt to assign a value meets a certain criteria. See “Property Get and Set Methods” on page 8-38.
- Trigger an event notification when any attempt is made to get or set its value. See “Property-Set and Query Events” on page 11-14.
- Control access by code to the property values. See the `SetAccess` and `GetAccess` attributes “Property Attributes” on page 8-8.
- Control whether its value is saved with the object in a MAT-file. See “Save and Load Objects” on page 13-2.

For an example of a class that defines and uses a class, see “Creating a Simple Class” on page 2-2.

Types of Properties

There are two types of properties:

- Stored properties — Use memory and are part of the object
- Dependent properties — No allocated memory and the get access method calculates the value when queried

Features of Stored Properties

- Property value is stored when you save the object to a MAT-file
- Can assign a default value in the class definition
- Can restrict property value to a specific class and size
- Can execute validation functions to control allowed property value (default and assigned)

- Can use a set access method to control possible values when set

When to Use Stored Properties

- You want to be able to save the property value in a MAT-file
- The property value is not dependent on other property values

Features of Dependent Properties

Dependent properties save memory because property values that depend on other values are calculated only when needed.

When to Use Dependent Properties

Define properties as dependent when you want to:

- Compute the value of a property from other values (for example, you can compute area from `Width` and `Height` properties).
- Provide a value in different formats depending on other values. For example, the size of a push button in values determined by the current setting of its `Units` property.
- Provide a standard interface where a particular property is or is not used, depending on other values. For example, different computer platforms can have different components on a toolbar).

For examples of classes that use dependent properties, see “Calculate Data on Demand” on page 3-17 and “A Class Hierarchy for Heterogeneous Arrays” on page 20-2.

See Also

Related Examples

- “Property Attributes” on page 8-8
- “Validate Property Values” on page 8-18
- “Property Get and Set Methods” on page 8-38
- “Static Properties” on page 5-38

Property Syntax

In this section...

This topic describes how to define class properties in MATLAB using `properties...end` blocks, and it introduces property validation syntax and concepts. It also covers the basics of getting and setting property values from a class instance.

Property Definition Block

The `properties` and `end` keywords define one or more class properties that have the same attribute settings. This is the general syntax for defining a property block:

```
properties (attributes)
   propName1
   ...
   propNameN
end
```

Note Properties cannot have the same name as the class or any of the other members defined by the class.

For example, this properties block defines two properties with the `SetAccess` attribute set to `private`. This attribute setting means that the property values can be set only by members of the `PrivateProps` class.

```
classdef PrivateProps
   properties (SetAccess = private)
      Property1
      Property2
   end
end
```

You can also define multiple property blocks for properties with different attributes. In this example, one properties block defines properties with private `SetAccess`, and the second block defines an abstract property. Property blocks with different attributes can appear in any order in the class definition.

```
classdef MultiplePropBlocks
   properties (SetAccess = private)
      Property1
      Property2
   end
   properties (Abstract)
      Property3
   end
end
```

For a full listing of property attributes, see “Property Attributes” on page 8-8.

Property Validation Syntax

Within a properties block, you can use property validation. Property validation enables you to place one or more restrictions on each property value, including size and class. You can also define a default value for each property. The general syntax for property validation is:

```
properties (attributes)
   propName1 (dimensions) class {validators} = defaultValue
   ...
end
```

- *`(dimensions)`* — Size of property value, specified as a comma-separated list of two or more numbers enclosed in parentheses, such as `(1,2)` or `(1,:)`. A colon allows any length in that dimension. The dimensions of the value must match *`(dimensions)`* exactly or be compatible. See “Compatible Array Sizes for Basic Operations” for more information. *`(dimensions)`* cannot include expressions.
- *`class`* — Class of property value, specified as the name of the class, such as `double`. The value must be the specified class or a class that can be converted. For example, a property that specifies `double` accepts values of type `single` and converts them to `double`. Beyond the classes already available in MATLAB, you can use your own classes as property validators. For user-defined classes, property validation allows a subclass of the specified *`class`* to pass without error, but it does not convert the subclass to the superclass.
- *`{validators}`* — Validation functions, specified as a comma-separated list enclosed in curly brackets, such as `mustBePositive` and `mustBeScalarOrEmpty`. Unlike *`class`*, validation functions do not modify property values. Validation functions error when the property values do not match their conditions. For a list of validation functions, see “Property Validation Functions” on page 8-29. You can also define your own validation functions.
- *`defaultValue`* — Default property values must conform to the specified size, class, and validation rules. A default value can also be an expression. See “Define Properties with Default Values” on page 8-13 for more information about how MATLAB evaluates default value expressions.

This class defines one property. The properties block has no explicit attribute defined, which is equivalent to defining a block of public properties. `MyPublicData` must also be a vector of positive doubles, and it has a default value of `[1 1 1]`.

```
classdef ValidationExample
   properties
      MyPublicData (1,:) double {mustBePositive} = [1 1 1]
   end
end
```

Not all validation options must be used at once, and different properties in the same block can use different combinations of validators. In this example, the `RestrictedByClass` property uses class validation only, while `RestrictedByFunction` uses a validation function and assigns a default value.

```
classdef DifferentValidation
   properties
      RestrictedByClass uint32
      RestrictedByFunction {mustBeInteger} = 0
```

```
   end
end
```

For more information, see “Property Class and Size Validation” on page 8-23 and “Property Validation Functions” on page 8-29.

Property Access Syntax

Property access syntax is like MATLAB structure field syntax. For example, if `obj` is an object of a class, then you can get the value of a property by referencing the property name.

```
val = obj.PropertyName
```

Assign values to properties by putting the property reference on the left side of the equal sign.

```
obj.PropertyName = val
```

For example, instantiate the `ValidationExample` class and read the value of `MyPublicData`.

```
classdef ValidationExample
   properties
      MyPublicData (1,:) double {mustBePositive} = [1 1 1]
   end
end

x = ValidationExample;
x.MyPublicData

ans =

     1     1     1
```

Assign a new value to the property that satisfies the validators defined for it.

```
x.MyPublicData = [2 3 5 7];
```

You can optionally define get and set methods that MATLAB automatically calls when you use this structure field syntax. For more information, see “Property Get and Set Methods” on page 8-38.

Reference Properties Using Variables

MATLAB can resolve a property name from a `string` or `char` variable using an expression of the form:

```
object.(PropertyNameVar)
```

`PropertyNameVar` is a variable containing the name of a valid object property. Use this syntax when passing property names as arguments. For example, the `getPropValue` function returns the value of the `KeyType` property.

```
PropName = "KeyType";
function o = getPropValue(obj,PropName)
   o = obj.(PropName);
end
```

See Also

Related Examples

- “Property Attributes” on page 8-8
- “Validate Property Values” on page 8-18
- “Initialize Property Values” on page 8-13

Property Attributes

In this section...

Purpose of Property Attributes

You can specify attributes in the class definition to customize the behavior of properties for specific purposes. Control characteristics like access, data storage, and visibility of properties by setting attributes. Subclasses do not inherit superclass member attributes.

Specify Property Attributes

Assign property attributes on the same line as the `properties` keyword.

```
properties (Attribute1 = value1, Attribute2 = value2,...)
   ...
end
```

For example, define a property `Data` with `private` access.

```
properties (Access = private)
   Data
end
```

You can use a simpler syntax for attributes whose values are `true`. The attribute name by itself implies true, and adding the not operator (`~`) to the name implies false. For example, this block defines abstract properties.

```
properties (Abstract)
   ...
end
```

Table of Property Attributes

All properties support the attributes listed in this table. Attribute values apply to all properties defined within the `properties...end` code block that specifies the nondefault values. Attributes that you do not explicitly defined take their default values.

Property Attributes

Attribute	Values	Additional Information
`AbortSet`	• `true` – MATLAB does not set the property value or call a set method if the new value is the same as the current value. • `false` (default) – MATLAB sets the property value regardless of the current value.	For handle classes only. Setting `AbortSet` to `true` also prevents the triggering of property `PreSet` and `PostSet` events. For more information, see "Assignment When Property Value Is Unchanged" on page 11-35.
`Abstract`	• `true` – The property has no implementation, but a concrete subclass must override this property without `Abstract` set to `true`. • `false` (default) – The property is concrete and does not need to be overridden in a subclass.	Abstract properties cannot define set or get access methods. See "Property Get and Set Methods" on page 8-38. Abstract properties cannot define initial values. A sealed class cannot define abstract members.
`Access`	• `public` (default) – The property can be accessed from any code. • `protected` – The property can be accessed from the defining class or its subclasses. • `private` – The property can be accessed only by members of the defining class. • List of classes that have get and set access to this property. Specify classes as a single `meta.class` object or a cell array of `meta.class` objects. See "Property Access Lists" on page 8-12 for more information.	Use `Access` to set `SetAccess` and `GetAccess` to the same value. Specifying `Access` as an empty cell array, `{}`, is the same as `private` access. See "Class Members Access" on page 12-23 for more information.

Attribute	Values	Additional Information
`Constant`	• `true` – The property has the same value in all instances of the class. • `false` (default) – The property value can vary between instances.	Subclasses inherit constant properties but cannot change them. Constant properties cannot also be defined as dependent. The value of `SetAccess` is ignored for constant properties. See “Define Class Properties with Constant Values” on page 15-2 for more information.
`Dependent`	• `true` – The property value is not stored in the object. The value is calculated when the property is accessed. • `false` (default) – The property value is stored in the object.	You can define set methods for dependent properties, but the set method cannot actually set the value of the property. It can take other actions, such as setting the value of another property. See “When to Use Set Methods with Dependent Properties” on page 8-43 for an example. Values returned by dependent property get methods are not considered when testing for object equality using `isequal`.
`GetAccess`	• `public` (default) – The property can be read from any code. • `protected` – The property can be read from the defining class or its subclasses. • `private` – The property can be read only by members of the defining class. • List of classes that can read this property. Specify classes as a single `meta.class` object or a cell array of `meta.class` objects. See “Property Access Lists” on page 8-12 for more information.	Specifying `GetAccess` as an empty cell array, `{}`, is the same as `private` access. In the Command Window, MATLAB does not display the names and values of properties with `protected` or `private` `GetAccess`. All subclasses must specify the same values as the superclass for the property `SetAccess` and `GetAccess` attributes. See “Class Members Access” on page 12-23 for more information.
`GetObservable`	• `true` – You can create listeners for handle class properties. The listeners are called whenever property values are queried. • `false` (default) – Listeners do not have access to this property.	See “Property-Set and Query Events” on page 11-14 for more information.

Attribute	Values	Additional Information
`Hidden`	• `true` – The property is not visible in property lists or in results from calls to `get`, `set`, or the `properties` functions. • `false` (default) – The property is visible.	In the Command Window, MATLAB does not display the names and values of properties whose `Hidden` attribute is `true`. However, hidden properties are visible in the Class Diagram Viewer app.
`NonCopyable`	• `true` – The property value is not copied when the object that defines it is copied. • `false` (default) – The property value is copied when the object is copied.	You can set `NonCopyable` to `true` only in handle classes. For more information, see “Exclude Properties from Copy” on page 7-33.
`PartialMatchPriority`	Positive integer – Defines the relative priority of partial property name matches used in `get` and `set` methods. The default value is 1.	Use only with subclasses of `matlab.mixin.SetGet`. For more information, see “Set Priority for Matching Partial Property Names” on page 7-27.
`SetAccess`	• `public` (default) – The property can be set from any code. • `protected` – The property can be set from the defining class or its subclasses. • `private` – The property can be set only by members of the defining class. • `immutable` – The property can be set only by the constructor. • List of classes that have set access to this property. Specify classes as a single `meta.class` object or a cell array of `meta.class` objects. See “Property Access Lists” on page 8-12 for more information.	All subclasses must specify the same values as the superclass for the property `SetAccess` and `GetAccess` attributes. For more information, see “Class Members Access” on page 12-23, “Properties Containing Objects” on page 8-45, and “Mutable and Immutable Properties” on page 8-16.
`SetObservable`	• `true` – You can create listeners for handle class properties. The listeners are called whenever property values are set. • `false` (default) – Listeners do not have access to this property.	See “Property-Set and Query Events” on page 11-14 for more information.

Attribute	Values	Additional Information
`Transient`	• `true` – The property value is not saved when the object is saved to a file or sent from MATLAB to another program, such as a MATLAB Engine application. • `false` (default) – The property value is saved when the object is saved.	See "Save and Load Process for Objects" on page 13-2 for more information.
Framework attributes	Classes that use certain framework base classes have framework-specific attributes. See the documentation for the specific base class you are using for information on these attributes.	

Property Access Lists

You can use lists of `meta.class` instances for the `Access`, `GetAccess`, and `SetAccess` attributes. For example, this class declares access lists for the `Prop1` and `Prop2` properties.

```
classdef PropertyAccess
   properties (GetAccess = {?ClassA, ?ClassB})
      Prop1
   end
   properties (Access = ?ClassC)
      Prop2
   end
end
```

For `Prop1`:

- Classes `ClassA` and `ClassB` have get access to `Prop1`.
- All subclasses of `ClassA` and `ClassB` have get access to `Prop1`.
- Access lists are not inherited, so subclasses of `PropertyAccess` do not have get access to `Prop1` unless they explicitly define that access.

For `Prop2`:

- `ClassC` has get and set access to `Prop2`.
- All subclasses of `ClassC` have get and set access to `Prop2`.
- Access lists are not inherited, so subclasses of `PropertyAccess` do not have access to `Prop2` unless they explicitly define that access.

See Also

Related Examples

- "Property Syntax" on page 8-4
- "Initialize Property Values" on page 8-13

Initialize Property Values

In this section...

There are two basic ways to initialize property values:

- **Define properties with default values** — MATLAB assigns the same initial value to the property of every instance.
- **Set property values in the constructor** — The constructor evaluates the assignment statement for each instance, which enables instances to have unique initial property values.

Define Properties with Default Values

You can assign a default value to an individual property using a value or an expression. Expressions cannot include variables. This example shows several ways to define a default value for a property.

```
classdef PropExample
   properties
      Prop1
      Prop2 = "some text"
      Prop3 = sin(pi/12)
      Prop4 = datetime.empty
      Prop5 (1,1) double {mustBePositive} = 1
   end
end
```

- `Prop1` — The property definition does not specify a default value, so MATLAB initializes the property value to an empty `double` (`[]`).
- `Prop2` — The default value is the string scalar `"some text"`.
- `Prop3` — The default value is the value of `sin(pi/12)`. Reading this property returns the evaluated expression (`0.2588`), not the expression itself.

  For more information on the evaluation of expressions that you assign as default values, see “Evaluation of Expressions in Class Definitions” on page 6-9 and “Properties Containing Objects” on page 8-45.
- `Prop4` — The default value is an empty `datetime` object.
- `Prop5` — The default value is `1`, and the property value in general is restricted to scalar, positive doubles. When a property definition specifies any size, class, or validation function restrictions on the property value, then the default value must satisfy those conditions. For example, a default value of 0 would cause an error during instantiation because it does not satisfy `mustBePositive`.

  For information on property restrictions based on size, class, and validation functions, see “Validate Property Values” on page 8-18.

Note MATLAB evaluates a default expression when the property value is first needed (for example, when the class is first instantiated). The same default value is then used for all instances of a class.

MATLAB does not reevaluate the default expression unless the class definition is cleared from memory.

Handle Objects as Default Property Values

When you use a handle class constructor to create a default property value, MATLAB calls the constructor only when the class is first used, and then uses the same object handle as the default for the property in all instances. Because all of the object handles reference the same object, any changes you make to the handle object in one instance are reflected in the handle object in all instances. To initialize a property value with a new instance of a handle object each time you instantiate your class, assign the property value in the constructor.

Set Property Values in the Constructor

To assign a value to a property from within the class constructor, refer to the object that the constructor returns (the output variable `obj`) and the property name using dot notation.

```
classdef MyClass
   properties
      Prop1
   end
   methods
      function obj = MyClass(intval)
         obj.Prop1 = intval;
      end
   end
end
```

When you assign a value to a property in the class constructor, MATLAB evaluates the assignment statement for each object you create. Assign property values in the constructor if you want each object to contain a unique value for that property.

For example, `ContainsHandle` assigns a unique handle object of class `MyHandleClass` to `Prop1` for each instance. `ContainsHandle` does this by calling the `MyHandleClass` constructor from its own constructor.

```
classdef ContainsHandle
   properties
       Prop1
   end
   methods
       function obj = ContainsHandle(keySet,valueSet)
           obj.Prop1 = MyHandleClass(keySet,valueSet);
       end
   end
end
```

For more information on constructor methods, see “Referencing the Object in a Constructor” on page 9-18.

Property Validation Before Construction

MATLAB validates default property values before the assignment of values in the constructor. The default value assigned in the `properties` block and any value set for the property in the class constructor must satisfy the specified validation. For example, `PropInit` restricts `Prop` to a scalar

positive double, but it does not assign a default value. By default, MATLAB assigns an initial value of empty double.

```
classdef PropInit
    properties
        Prop (1,1) double {mustBePositive}
    end
    methods
        function obj = PropInit(positiveInput)
            obj.Prop = positiveInput;
        end
    end
end
```

Calling the class constructor with a valid value for `Prop` still causes an error because of the initial empty double in `Prop`. An empty double does not satisfy the validation function `mustBePositive`.

```
obj = PropInit(2);
```

```
Error using implicit default value of property 'Prop' of class 'PropInit':
Value must be positive.
```

To avoid this problem, ensure that your properties have default values that satisfy your validation, even when you intend to overwrite those values in the constructor.

See Also

Related Examples

- “Evaluation of Expressions in Class Definitions” on page 6-9
- “Ways to Use Properties” on page 8-2
- “Validate Property Values” on page 8-18

Mutable and Immutable Properties

In this section...

Set Access to Property Values

The property `SetAccess` attribute enables you to determine under what conditions code can modify object property values. There are four levels of set access that provide varying degrees of access to object property values:

- `SetAccess = public` — Any code with access to an object can set public property values. There are differences between the behavior of handle and value classes with respect to modifying object properties.
- `SetAccess = protected` — Only code executing from within class methods or methods of subclasses can set property values. You cannot change the value of an object property unless the class or any of its subclasses defines a method to do so.
- `SetAccess = private` — Only the defining class can set property values. You can change the value of an object property only if the class defines a method that sets the property.
- `SetAccess = immutable` — Property value is set during construction. You cannot change the value of an immutable property after the object is created. Set the value of the property as a default or in the class constructor. You cannot define a property set method (`set.`*`PropertyName`*) for an immutable property.

For related information, see “Properties Containing Objects” on page 8-45.

Define Immutable Property

In this class definition, only the `Immute` class constructor can set the value of the `CurrentDate` property:

```
classdef Immute
   properties (SetAccess = immutable)
      CurrentDate
   end
   methods
      function obj = Immute
         obj.CurrentDate = datetime("today");
      end
   end
end

a = Immute

a =

  Immute with properties:

    CurrentDate: 09-Jun-2022
```

See Also

Related Examples

- “Property Attributes” on page 8-8
- “Object Modification” on page 5-35

Validate Property Values

In this section...

Property Validation in Class Definitions

MATLAB property validation enables you to place specific restrictions on property values. You can use validation to constrain the class and size of property values. Also, you can use functions to establish criteria that the property value must conform to. MATLAB defines a set of validation functions and you can write your own validation functions.

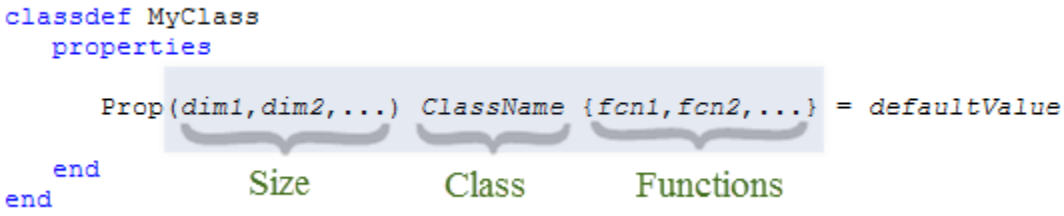
The use of property validation is optional in class definitions.

Additional Information on Property Validation

For more information on property validation, see “Property Class and Size Validation” on page 8-23, “Property Validation Functions” on page 8-29, and “Metadata Interface to Property Validation” on page 8-36.

Validation Syntax

The highlighted area in the following code shows the syntax for property validation.



Property validation includes any of the following:

- Size — The length of each dimension, specified as a positive integer or a colon. A colon indicates that any length is allowed in that dimension. The value assigned to the property must conform to the specified size or be compatible with the specified size. For more information, see “Property Size Validation” on page 8-23.
- Class — The name of a single MATLAB class. The value assigned to the property must be of the specified class or convertible to the specified class. Use any MATLAB class or externally defined class that is supported by MATLAB, except for Java and COM classes. For more information, see “Property Class Validation” on page 8-24.

- Functions — A comma-separated list of validation function names. MATLAB passes the value assigned to the property to each the validation functions after applying any possible class and size conversions. Validator functions throw errors if the validation fails, but do not return values. For more information, see “Property Validation Functions” on page 8-29.

  For a list of MATLAB validation functions, see “Property Validation Functions” on page 8-29.

Using Property Validation

Use property validation for public properties to control the values user code assigns to the properties.

If you want to restrict property values to a fixed set of identifiers, create an enumeration class for these identifiers and constrain the property to this class. For information on enumeration classes, see “Define Enumeration Classes” on page 14-4.

MATLAB type conversion rules apply to property validation. For example, MATLAB can coerce from one to another numeric type. Therefore, restricting a property value to a specific numeric type, such as double does not prevent the assignment of other numeric types to the property.

To ensure that a property can be assigned only a specific type of value, restrict the property to a type that supports only the desired type conversions or use a validation function to specify the exact class allowed for the property instead of specifying the property type. MATLAB evaluates the type specification before executing any validation functions. For more information, see “Order of Validation” on page 8-20.

Specify Valid Default

Ensure that any default value assigned to the property meets the restrictions imposed by the specified validation. If you do not specify a default value, MATLAB creates a default value by assigning an empty object of the specified class or by calling the default constructor if size restriction does not allow the use of an empty default value. The default constructor must return an object of the correct size.

Sample Class Using Property Validation

The `ValidateProps` class defines three properties with validation.

```
classdef ValidateProps
   properties
      Location(1,3) double {mustBeReal, mustBeFinite}
      Label(1,:) char {mustBeMember(Label,{'High','Medium','Low'})} = 'Low'
      State(1,1) matlab.lang.OnOffSwitchState
   end
end
```

- `Location` must be a 1-by-3 array of class `double` whose values are real, finite numbers.
- `Label` must be a `char` vector that is either `'High'`, `'Medium'`, or `'Low'`.
- `State` must be an enumeration member of the `matlab.lang.OnOffSwitchState` class (`off` or `on`).

Validation at Instantiation

Creating an object of the `ValidateProps` class performs the validation on implicit and explicit default values:

```
a = ValidateProps
```

```
a =

  ValidateProps with properties:

    Location: [0 0 0]
       Label: 'Low'
       State: off
```

When creating the object, MATLAB:

- Initializes the `Location` property value to `[0 0 0]` to satisfy the size and class requirements.
- Sets the `Label` property to its default value, `'Low'`. The default value must be a member of the allowed set of values. The empty `char` implicit default value would cause an error.
- Sets the `State` property to the `off` enumeration member defined by the `matlab.lang.OnOffSwitchState` class.

For information on how MATLAB selects default values, see “Default Values Per Size and Class” on page 8-28.

Order of Validation

When a value is assigned to the property, including default values that are specified in the class definition, MATLAB performs validation in this order:

- Class validation — This validation can cause conversion to a different class, such as conversion of a `char` to `string`. Assignment to properties follows MATLAB conversion rules for arrays.
- Size validation — This validation can cause size conversion, such as scalar expansion or conversion of a column vector to a row vector. Assignment to a property that specifies a size validation behaves the same as assignment to any MATLAB array. For information on indexed assignment, see “Array Indexing”.
- Validator functions — MATLAB passes the result of the class and size validation to each validation function, in left to right order. An error can occur before all validation functions have been called, which ends the validation process.
- Set method — MATLAB performs property validation before calling a property set method, if one is defined for that property. Assignment to the property within a property set or get method does not apply the validation again. Often, you can replace property set methods using property validation.

Property Validation Errors

The `ValueProp` class uses size, class, and function validation to ensure that an assignment to the `Value` property is a double scalar that is not negative.

```
classdef ValueProp
   properties
      Value(1,1) double {mustBeNonnegative} = 0
   end
end
```

This statement attempts to assign a cell array to the property. This assignment violates the class validation.

```
a.Value = {10,20};
```

```
Error setting property 'Value' of class 'ValueProp':
Invalid data type. Value must be double or be convertible to double.
```

This statement attempts to assign a 1-by-2 double array to the property. This assignment violates the size validation.

```
a.Value = [10 20];
```

```
Error setting property 'Value' of class 'ValueProp':
Size of value must be scalar.
```

This statement attempts to assign a scalar double to the property. This assignment fails the function validation, which requires a nonnegative number.

```
a.Value = -10;
```

```
Error setting property 'Value' of class 'ValueProp':
Value must be nonnegative.
```

The validation process ends with the first error encountered.

Abstract Property Validation

You can define property validation for abstract properties. The validation applies to all subclasses that implement the property. However, subclasses cannot use any validation on their implementation of the property. When inheriting validation for a property from multiple classes, only a single abstract property in one superclass can define the validation. None of the superclasses can define the property as concrete.

Objects Not Updated When Changing Validation

If you change the property validation while objects of the class exist, MATLAB does not attempt to apply the new validation to existing property values. However, MATLAB does apply the new validation when you make assignments to the properties of existing objects.

Validation During Load Operation

When saving an object to a `MAT` file, MATLAB saves all nondefault property values with the object. When loading the object, MATLAB restores these property values in the newly created object.

If a class definition changes the property validation such that the loaded property value is no longer valid, MATLAB substitutes the currently defined default value for that property. However, the `load` function suppresses the validation errors that occur before assigning the default value from the current class definition. Therefore, validation errors are silently ignored during load operations.

To illustrate this behavior, this example creates, saves, and loads an object of the `MonthTemp` class. This class restricts the `AveTemp` property to a cell array.

```
classdef MonthTemp
   properties
      AveTemp cell
   end
end
```

Create a `MonthTemp` object and assign a value to the `AveTemp` property.

```
a = MonthTemp;
a.AveTemp = {'May',70};
```

Save the object using `save`.

```
save TemperatureFile a
```

Edit the property definition to change the validation class for the `AveTemp` property from cell array to `containers.Map`.

```
classdef MonthTemp
   properties
      AveTemp containers.Map
   end
end
```

Load the saved object with the new class definition on the MATLAB path. MATLAB cannot assign the saved value to the `AveTemp` property because the cell array, `{'May',70}`, is not compatible with the current requirement that the property value be a `containers.Map` object. MATLAB cannot convert a cell array to a `containers.Map`.

To address the incompatibility, MATLAB sets the `AveTemp` property of the loaded object to the current default value, which is an empty `containers.Map` object.

```
load TemperatureFile a
 a.AveTemp

ans =

  Map with properties:

       Count: 0
     KeyType: char
   ValueType: any
```

The loaded object has a different value assigned to the `AveTemp` property because the saved value is now invalid. However, the load process suppresses the validation error.

To prevent loss of data when changing class definitions and reloading objects, implement a `loadobj` method or class converter method that enables the saved values to satisfy the current property validation.

For more information on saving and loading objects, see “Save and Load Process for Objects” on page 13-2.

See Also

Related Examples

- “Property Class and Size Validation” on page 8-23
- “Property Validation Functions” on page 8-29

Property Class and Size Validation

In this section...

Property Class and Size

MATLAB applies any class and size validation defined for a property before calling validation functions. Assignment to a property that defines size or class validation is analogous to assignment to a MATLAB object array. MATLAB can apply class and size conversions to the right side of the assignment to satisfy class and size validation.

For more information, see “Order of Validation” on page 8-20 and “Property Validation Functions” on page 8-29.

Property Size Validation

Specify the property size as the row, column, and additional dimension following the property name.

```
classdef MyClass
   properties
      Prop(dim1,dim2,...) = defaultValue
   end
end
```

Assignment and Size Validation

This class defines the size of the `Location` property as 1-by-3. Any value assigned to this property must conform to that size or must be convertible to that size.

```
classdef ValidateProps
   properties
      Location(1,3)
   end
end
```

The implicit default value assigned by MATLAB, `[0 0 0]`, conforms to the specified size:

```
a = ValidateProps

a =

  ValidateProps with properties:

    Location: [0 0 0]
```

MATLAB applies scalar expansion when you assign a scalar the `Location` property.

```
a = ValidateProps;
a.Location = 1
```

```
a =

  ValidateProps with properties:

    Location: [1 1 1]
```

MATLAB converts columns to rows to match the size specification:

```
col = [1;1;1]
```

```
col =

     1
     1
     1
```

```
a.Location = col
```

```
a =

  ValidateProps with properties:

    Location: [1 1 1]
```

Colon in Size Specification

A colon in the size specification indicates that the corresponding dimension can have any length. For example, you can assign a value of any length to the `Label` property in this class.

```
classdef ValidateProps
   properties
      Label(1,:)
   end
end
```

```
a = ValidateProps;
a.Label = 'Click to Start'
```

```
a =

  ValidateProps with properties:

    Label: 'Click to Start'
```

Assignment to a property that defines size validation follows the same rules as the equivalent indexed array assignment. For information on indexing behavior of multidimensional arrays, see “Compatible Array Sizes for Basic Operations”.

Property Class Validation

Defining the class of a property can reduce the need to test the values assigned to the property in your code. In general, the value assigned to the property must be of the specified class or convertible to the specified class. An exception to this rule is for subclasses. If you specify a user-defined class as part of validation, subclasses of that class pass validation without error, but they are not converted.

You can specify only one class per property. Use validation functions like `mustBeNumeric` or `mustBeInteger` to restrict properties to a broader category of classes. For more information on validation functions, see “Property Validation Functions” on page 8-29.

You can use any MATLAB class or externally defined class that is supported by MATLAB, except Java and COM classes.

Place the name of the class in the property definition block following the property name and optional size specification.

```
classdef MyClass
   properties
      Prop ClassName = defaultValue
   end
end
```

If you do not specify a default value, MATLAB assigns an empty object of the specified class to the property. If you define a size and a class, MATLAB attempts to create a default value for the property that satisfies both the size and class requirement.

MATLAB creates the default value by calling the class constructor with no arguments. The class must have a constructor that returns an object of the specified size when called with no input arguments or you must specify a default value for the property that satisfies the property size restriction. For more information, see “Default Values Per Size and Class” on page 8-28.

Using Class Validation

The `PropsWithClass` class defines two properties with class definitions:

- `Number` — Values must be of class `double` or convertible to `double`.
- `Today` — Values must be of class `string` or convertible to `string`. The default value is the current date, returned by the `datetime` function.

```
classdef PropsWithClass
   properties
      Number double
      Today string = datetime("today")
   end
end
```

Create an object of the `PropsWithClass` class.

```
p = PropsWithClass
```

```
p =

  PropsWithClass with properties:

    Number: []
     Today: "09-Jun-2022"
```

MATLAB performs conversions from any compatible class to the property class. For example, assign a `unint8` value to the `Number` property.

```
p.Number = uint8(5);
disp(class(p.Number))
```

```
ans =

double
```

Because the `uint8` value can be converted to `double`, you can assign that `uint8` value to the `Number` property.

Assigning an incompatible value to a property that uses class validation causes an error.

```
p.Number = datetime("today");
```

```
Error setting property 'Number' of class 'PropsWithClass':
Value must be double or be convertible to double.
```

User-Defined Class for Validation

You can define a class to control the values assigned to a property. Enumeration classes enable users to set property values to character vectors or string scalars with inexact name matching.

For example, suppose that there is a class that represents a three-speed mechanical pump. You can define an enumeration class to represent the three flow rates.

```
classdef FlowRate < int32
   enumeration
      Low    (10)
      Medium (50)
      High   (100)
   end
end
```

The `Pump` class has a method to return the current flow rate in gallons per minute. Define the `Speed` property as a `FlowRate` class.

```
classdef Pump
   properties
      Speed FlowRate
   end
   methods
      function getGPM(p)
         if isempty(p.Speed)
            gpm = 0;
         else
            gpm = int32(p.Speed);
         end
         fprintf('Flow rate is: %i GPM\n',gpm);
      end
   end
end
```

Users can set the `Speed` property using inexact text.

```
p = Pump;
p.Speed = 'm'
```

```
p =

  Pump with properties:

    Speed: Medium
```

The numerical value is available from the property.

```
getGPM(p)
```

```
Flow rate is: 50 GPM
```

For information about enumeration classes, see “Define Enumeration Classes” on page 14-4.

Integer Class Validation

MATLAB supports several integer classes (see “Integers”). However, restricting a property to an integer class can result in integer overflow. The resulting value can saturate at the maximum or minimum value in the integer’s range.

When integer overflow occurs, the value that is assigned to a property can be a value that is different from the value from the right side of the assignment statement.

For example, suppose that you want to restrict a property value to a scalar `uint8`.

```
classdef IntProperty
   properties
      Value(1,1) uint8
   end
end
```

Assigning a numeric value to the `Value` property effectively casts the numeric value to `uint8`, but does not result in an error for out-of-range values.

```
a = IntProperty;
a.Value = -10;
disp(a.Value)
```

```
0
```

Assignment to the `Value` property is equivalent to indexed assignment of an array. If the assigned value is out of the range of values that `uint8` can represent, MATLAB sets the value to the closest value that it can represent using `uint8`.

```
a = uint8.empty;
a(1) = -10
```

```
a =

  uint8

   0
```

To avoid the potential for integer overflow, use a combination of validation functions that restrict the value to the intended range instead of an integer class.

```
classdef IntProperty
   properties
      Value(1,1) {mustBeInteger, mustBeNonnegative,...
         mustBeLessThan(Value,256)}
   end
end
```

Because there is no conversion of the assigned value by the `uint8` class, the validators catch out of range values and throw an appropriate error.

```
a = IntProperty;
a.Value = -10;
```

```
Error setting property 'Value' of class 'IntProperty':
Value must be nonnegative.
```

Default Values Per Size and Class

Any default property value that you assign in the class definition must conform to the specified validation.

Implicit Default Values

MATLAB defines a default value implicitly if you do not specify a default value in the class definition. This table shows how size and class determine the implicit default value of MATLAB classes.

Size	Class	Implicit Default Assigned by MATLAB
`(m,n)`	Any numeric	m-by-n array of zeros of specified class.
`(m,:)` or `(:,n)`	Any class	m-by-0 or 0-by-n of specified class.
`(m,n)`	`char`	m-by-n char array of spaces.
`(m,n)`	`cell`	m-by-n cell array with each cell containing a 0-by-0 double.
`(m,n)`	`struct`	m-by-n `struct`
`(m,n)`	`string`	m-by-n `string`
`(m,n)`	enumeration class	First enumeration member defined in the class.
`(1,1)`	`function_handle`	Runtime error — define a default value in the class.

To determine the implicit default value for nonzero and explicit size specifications, MATLAB calls the default class constructor and builds an array of the specified size using the instance returned by the constructor call. If the class does not support a default constructor (that is, a constructor called with no arguments), then MATLAB throws an error when instantiating the class containing the validation.

If the specified size has any zero or unrestricted (`:`) dimensions, MATLAB creates a default value that is an empty array with the unrestricted dimension set to zero.

For heterogeneous arrays, MATLAB calls the `matlab.mixin.Heterogeneous.getDefaultScalarElement` method to get the default object.

See Also

Related Examples

- “Validate Property Values” on page 8-18
- “Property Validation Functions” on page 8-29
- “Enumerations for Property Values” on page 14-14

Property Validation Functions

In this section...

MATLAB Validation Functions

MATLAB defines functions for use in property validation. These functions support common use patterns for validation and provide descriptive error messages. The following tables categorize the MATLAB validation functions and describe their use.

Numeric Value Attributes

Name	Meaning	Functions Called on Inputs
`mustBePositive(value)`	`value > 0`	`gt`, `isreal`, `isnumeric`, `islogical`
`mustBeNonpositive(value)`	`value <= 0`	`ge`, `isreal`, `isnumeric`, `islogical`
`mustBeNonnegative(value)`	`value >= 0`	`ge`, `isreal`, `isnumeric`, `islogical`
`mustBeNegative(value)`	`value < 0`	`lt`, `isreal`, `isnumeric`, `islogical`
`mustBeFinite(value)`	`value` has no `NaN` and no `Inf` elements.	`isfinite`
`mustBeNonNan(value)`	`value` has no `NaN` elements.	`isnan`
`mustBeNonzero(value)`	`value ~= 0`	`eq`, `isnumeric`, `islogical`
`mustBeNonsparse(value)`	`value` has no sparse elements.	`issparse`
`mustBeReal(value)`	`value` has no imaginary part.	`isreal`
`mustBeInteger(value)`	`value == floor(value)`	`isreal`, `isfinite`, `floor`, `isnumeric`, `islogical`
`mustBeNonmissing(value)`	`value` cannot contain missing values.	`ismissing`

Comparison with Other Values

Name	Meaning	Functions Called on Inputs
`mustBeGreaterThan(value,c)`	`value > c`	`gt`, `isreal`, `isnumeric`, `islogical`
`mustBeLessThan(value,c)`	`value < c`	`lt`, `isreal`, `isnumeric`, `islogical`
`mustBeGreaterThanOrEqual(value,c)`	`value >= c`	`ge`, `isreal`, `isnumeric`, `islogical`
`mustBeLessThanOrEqual(value,c)`	`value <= c`	`le`, `isreal`, `isnumeric`, `islogical`

Data Types

Name	Meaning	Functions Called on Inputs
`mustBeA(value,classnames)`	`value` must be of specific class.	Uses class definition relationships
`mustBeNumeric(value)`	`value` must be numeric.	`isnumeric`
`mustBeNumericOrLogical(value)`	`value` must be numeric or logical.	`isnumeric`, `islogical`
`mustBeFloat(value)`	`value` must be floating-point array.	`isfloat`
`mustBeUnderlyingType(value,typename)`	`value` must have specified underlying type.	`isUnderlyingType`

Size

Name	Meaning	Functions Called on Inputs
`mustBeNonempty(value)`	`value` is not empty.	`isempty`
`mustBeScalarOrEmpty(value)`	`value` must be a scalar or be empty.	`isscalar`, `isempty`
`mustBeVector(value)`	`value` must be a vector.	`isvector`

Membership and Range

Name	Meaning	Functions Called on Inputs
`mustBeMember(value,S)`	`value` is an exact match for a member of S.	`ismember`
`mustBeInRange(value,lower,upper,boundflags)`	`value` must be within range.	`gt`, `ge`, `lt`, `le`

Text

Name	Meaning	Functions Called on Inputs
`mustBeFile(path)`	`path` must refer to a file.	`isfile`
`mustBeFolder(folder)`	`path` must refer to a folder.	`isfolder`
`mustBeNonzeroLengthText(value)`	`value` must be a piece of text with nonzero length.	Not applicable
`mustBeText(value)`	`value` must be a string array, character vector, or cell array of character vectors.	Not applicable
`mustBeTextScalar(value)`	`value` must be a single piece of text.	Not applicable
`mustBeValidVariableName(varname)`	`varname` must be a valid variable name.	`isvarname`

Validate Property Using Functions

Use property validation functions in class definitions to impose specific restrictions on property values. A validation function accepts a potential property value as an argument and issues an error if the value does not meet the specific requirement imposed by the function.

During the validation process, MATLAB passes the value to each validation function listed in the class definition. MATLAB calls each function from left to right and throws the first error encountered. The value passed to the validation functions is the result of any conversion applied by the class and size specifications. For more information on class and size validation, see “Property Class and Size Validation” on page 8-23.

For a list of MATLAB validation functions, see “MATLAB Validation Functions” on page 8-29.

Validation Function Syntax

Specify validation functions as a comma-separated list of function names or function calls with arguments, enclosed in braces.

```
classdef MyClass
   properties
```

```
      Prop {fcn1,fcn2} = defaultValue
   end
end
```

MATLAB passes the potential property value to the validation function implicitly. However, if the validation function requires input arguments in addition to the potential property value, then you must include both the property and the additional arguments. Additional arguments must be literal values and cannot reference variables. Literal values are nonsymbolic representations, such as numbers and text.

For example, consider the function `mustBeGreaterThan`. It requires a limiting value as an input parameter. This validation function requires that a property value must be greater than this limiting value.

Pass the property as the first argument. Use the property name, but do not enclose the name in quotation marks. This property definition restricts `Prop` to values greater than `10`.

```
properties
   Prop {mustBeGreaterThan(Prop,10)}
end
```

Using Validation Functions

The following class specifies validation functions for each property.

- `Data` must be numeric and finite.
- `Interp` must be one of the three options listed. Specify a default value for this property to satisfy this requirement.

```
classdef ValidatorFunction
   properties
      Data {mustBeNumeric, mustBeFinite}
      Interp {mustBeMember(Interp,{'linear','cubic','spline'})} = 'linear'
   end
end
```

Creating a default object of the class shows the initial values.

```
a = ValidatorFunction
```

```
a =

  ValidatorFunction with properties:

      Data: []
    Interp: 'linear'
```

Assigning values to properties calls the validation functions.

```
a.Data = 'cubic'
```

```
Error setting property 'Data' of class 'ValidatorFunction':
Value must be numeric.
```

Because the `Data` property validation does not include a numeric class, there is no conversion of the `char` vector to a numeric value. If you change the validation of the `Data` property to specify the class as `double`, MATLAB converts the `char` vector to a `double` array.

```
properties
   Data double {mustBeNumeric, mustBeFinite}
end
```

The assignment to the `char` vector does not produce an error because MATLAB converts the `char` vector to class `double`.

```
a.Data = 'cubic'

a =

  ValidatorFunction with properties:

      Data: [99 117 98 105 99]
    Interp: 'linear'
```

Assignment to the `Interp` property requires an exact match.

```
a = ValidatorFunction;
a.Interp = 'cu'

Error setting property 'Interp' of class 'ValidatorFunction':
Value must be a member of this set
    linear
    cubic
    spline
```

Using an enumeration class provides inexact matching and case insensitivity.

Enumeration Class for Inexact Matching

Property validation using an enumeration class provides these advantages:

- Inexact, case-insensitive matching for unambiguous char vectors or string scalars
- Conversion of inexact matches to correct values

For example, suppose that you define the `InterpMethod` enumeration class for the `Interp` property validation.

```
classdef InterpMethod
   enumeration
      linear
      cubic
      spline
   end
end
```

Change the `Interp` property validation to use the `InterpMethod` class.

```
classdef ValidatorFunction
   properties
      Data {mustBeNumeric, mustBeFinite}
      Interp InterpMethod
   end
end
```

Assign a value matching the first few letters of `'cubic'`.

```
a = ValidatorFunction;
a.Interp = 'cu'

a =

  ValidatorFunction with properties:

      Data: []
    Interp: cubic
```

Define Validation Functions

Validation functions are ordinary MATLAB functions that are designed for the specific purpose of validating property and function argument values. Functions used to validate properties:

- Accept the potential property value as an input argument
- Do not return values
- Display error messages if the validation fails

Creating your own validation function is useful when you want to provide specific validation that is not available using the MATLAB validation functions. You can create local functions within the class file or place the function on the MATLAB path to be available for use in any class.

For example, the `ImgData` class uses a local function to define a validator that restricts the `Data` property to only `uint8` or `uint16` values, excluding subclasses and not allowing conversion from other numeric classes. The predefined validation function `mustBeInRange` restricts the range of allowed values.

```
classdef ImgData
    properties
        Data {mustBeImData(Data),mustBeInRange(Data,0,255)} = uint8(0)
    end
end
function mustBeImData(a)
    if ~(isa(a,'uint8') || isa(a,'uint16'))
        error("Value of Data property must be uint8 or uint16 data.")
    end
end
```

When you create an instance of the `ImgData` class, MATLAB validates that the default value is a `uint8` or `uint16` value, in the range `0...255`, and not empty. Note that the default value must satisfy the validation requirements like any other value assigned to the property.

```
a = ImgData

a =

  ImgData with properties:

    Data: 0
```

Property assignment invokes the validators in left-to-right order. Assigning a `char` vector to the `Data` property causes an error thrown by `mustBeImData`.

```
a.Data = 'red';

Error setting property 'Data' of class 'ImgData'.
Value of Data property must be uint8 or uint16 data.
```

Assigning a numeric value that is out of range causes an error thrown by `mustBeInRange`.

```
a.Data = uint16(312);
```

```
Error setting property 'Data' of class 'ImgData'. Value must be greater
than or equal to 0, and less than or equal to 255.
```

For related functions, see `mustBeInteger`, `mustBeNumeric`, and `mustBePositive`.

See Also

Related Examples

- “Validate Property Values” on page 8-18
- “Property Class and Size Validation” on page 8-23

Metadata Interface to Property Validation

For information on property validation, see “Validate Property Values” on page 8-18.

You can determine what validation applies to a property by accessing the validation metadata. Instances of the `meta.Validation` class provide the following information about property validation.

- Class requirement of the property specified as a `meta.class` object
- Size requirements of the property value specified as an array of `meta.FixedDimension` and `meta.UnrestrictedDimension` objects
- Function handles referencing validation functions applied to property values specified as a cell array of function handles.

For example, the `ValidationExample` class defines a property that must be an array of doubles that is 1-by-any number of elements and must be a real number that is greater than 10.

```
classdef ValidationExample
   properties
      Prop (1,:) double {mustBeReal, mustBeGreaterThan(Prop, 10)} = 200;
   end
end
```

Access the `meta.Validation` object from the property's `meta.property` object. Get the validation information from the `meta.Validation` object properties. Collection this information into a cell array.

- Get the size information from the `Size` property
- Get the class name from the `Class` property
- Get a cell array of function handles for the validation functions from the `ValidatorFunctions` property.

```
mc = ?ValidationExample;
mp = findobj(mc.PropertyList,'Name','Prop');
sz = mp.Validation.Size;
len = length(sz);
dim = cell(1:len);
   for k = 1:len
      switch class(sz(k))
         case 'meta.FixedDimension'
            dim{k} = sz(k).Length;
         case 'meta.UnrestrictedDimension'
            dim{k} = ':';
      end
   end
dim{end+1} = mp.Validation.Class.Name;
dim{end+1} = mp.Validation.ValidatorFunctions;
```

See Also

`meta.property` | `meta.Validation`

Related Examples

- “Validate Property Values” on page 8-18
- “Property Class and Size Validation” on page 8-23
- “Property Validation Functions” on page 8-29

Property Get and Set Methods

In this section...

You can define property get and set methods that MATLAB calls automatically whenever the associated property is accessed. To associate a get or set method with a given property, name the get and set methods using the forms `get.`*`PropertyName`* and `set.`*`PropertyName`*, respectively.

Get and set methods can perform extra steps beyond just accessing the property. Use get methods to:

- Calculate the value of dependent properties.
- Store data in a different format than what you present to users.

Use set methods to:

- Design property validation that is more complex than what the built-in validation techniques support.
- Issue custom error messages.
- Perform actions that are a direct result of a property value change, such as establishing or updating connections with hardware devices or opening files, ensuring access to resources.

Get and set methods do add overhead to your classes. Avoid complex and computation-heavy operations in the get and set methods of frequently accessed properties.

Note **You cannot call the get and set methods described in this topic directly**. MATLAB automatically calls these methods when you access property values. For information on implementing user-callable get and set methods, see “Implement Set/Get Interface for Properties” on page 7-22.

Property Get Methods

You can define a get method that MATLAB automatically calls the whenever the associated property value is queried. The get method must return the property value. Get methods use this syntax, where *`PropertyName`* is the name of the property.

```
methods
   function value = get.PropertyName(obj)
      ...
   end
end
```

Method blocks defining get or set methods cannot specify attributes.

For example, the `triangleArea` class defines a get method for the `Area` property. `Area` is defined as a dependent property, which means that it does not store values. The get method for `Area` calculates the value on demand. (For more information on dependent properties, see “Get and Set Methods for Dependent Properties” on page 8-42.)

```
classdef triangleArea
   properties
```

```
      Base = 1
      Height = 1
   end
   properties (Dependent)
      Area
   end
   methods
      function a = get.Area(obj)
         disp("Executing get.Area method.")
         a = 0.5*obj.Base*obj.Height;
      end
   end
end
```

Create an instance of `triangleArea`.

```
a = triangleArea
```

```
a =

Executing get.Area method.
  triangleArea with properties:

      Base: 1
    Height: 1
      Area: 0.5000
```

When displaying an object, MATLAB calls any defined get methods for the properties it displays. In this case, it calls `get.Area` and calculates the value of `Area` based on the default values for `Base` and `Height`. If a get method errors, MATLAB suppresses the error and omits that property from the display.

Change the values of `Base` and `Height` and access `Area` again.

```
a.Base = 3;
a.Height = 4;
a.Area
```

```
Executing get.Area method.

ans =

     6
```

Get Method Usage

- Get methods are not called recursively.
- When copying a value object (that is, not derived from the `handle` class), get methods are not called when copying property values from one object to another.

Property Set Methods

You can define a set method that MATLAB automatically calls whenever the associated property is assigned a value. Set methods use these syntaxes, depending on whether the class is a value or handle class:

- Value class set methods must return the modified object.

```
methods
   function obj = set.PropertyName(obj,value)
      ...
   end
end
```

- Handle class set methods do not need to return the modified object.

```
methods
   function set.PropertyName(obj,value)
      ...
   end
end
```

Method blocks defining get or set methods cannot specify attributes.

For example, `symPosDef` uses a set method for property validation. When the `inputMatrix` property is set to a new value, the set method calls the `chol` function to determine if the input matrix is symmetric positive definite. If it is, the method sets `inputMatrix` to that value. If not, the method returns a custom error message.

```
classdef symPosDef
  properties
     inputMatrix = [1 0; 0 1]
  end
  methods
    function obj = set.inputMatrix(obj,val)
      try chol(val)
         obj.inputMatrix = val;
      catch ME
         error("inputMatrix must be symmetric positive definite.")
       end
     end
  end
end
```

Create an instance of `symPosDef` and try to set `inputMatrix` to a value that is not a symmetric positive definite matrix.

```
s = symPosDef;
s.inputMatrix = [1 2; 1 1]
```

```
Error using symPosDef/set.inputMatrix
inputMatrix must be symmetric positive definite.
```

Set Method Usage

- Set methods are not called recursively.
- MATLAB does not call set methods when it assigns default values to the properties during initialization of an object. However, setting property values in the constructor does call set methods.
- MATLAB calls set methods when an object is loaded.
- When MATLAB copies a value object (any object that is not a `handle`), set methods are not called when copying property values from one object to another.

- When a property is defined with the `AbortSet` attribute equal to `true`, the set method of the property is not called when assigning a value that is the same as the current value. However, if the property has a get method, that method is called so that the values can be compared. See “Assignment When Property Value Is Unchanged” on page 11-35 for more information on this attribute.

See Also

Related Examples

- “Get and Set Methods for Dependent Properties” on page 8-42
- “Listen for Changes to Property Values” on page 11-32

Get and Set Methods for Dependent Properties

In this section...

Dependent properties do not store data. The value of a dependent property depends on other values, such as the values of nondependent properties. Define a dependent property using this syntax:

```
properties (Dependent)
   PropertyName
end
```

Because dependent properties do not store data, you must define get methods (`get.`*`PropertyName`*) to determine the value for the properties when the properties are queried.

Dependent properties can also have set methods (`set.`*`PropertyName`*), but these methods cannot actually set the value of the dependent property. However, a set method can contain other code. For example, it can set values of properties related to the dependent property.

For an introduction to defining get and set methods, see “Property Get and Set Methods” on page 8-38.

Define a Get Method for a Dependent Property

The `Account` class stores an amount in US dollars, and it can return that value in one of three currencies: US dollars, euros, or Japanese yen. That converted value is represented by the dependent `Balance` property. The `get.Balance` method uses `USDollarAmount` and `Currency` to determine a conversion rate to calculate the `Balance` property.

```
classdef Account
   properties
      Currency {mustBeMember(Currency,["USD","EUR","JPY"])} = "USD"
      USDollarAmount = 0
   end
   properties (Dependent)
      Balance
   end
   methods
      function value = get.Balance(obj)
         c = obj.Currency;
         switch c
            case "EUR"
               v = obj.USDollarAmount/0.98;
            case "JPY"
               v = obj.USDollarAmount/0.0069;
            otherwise
               v = obj.USDollarAmount;
         end
         value = v;
      end
   end
end
```

Create an instance of `Account`. Set the `USDollarAmount` and `Currency` properties.

```
a = Account;
a.USDollarAmount = 100;
a.Currency = "JPY";
```

You cannot explicitly call get methods. When you access `Balance`, MATLAB calls the get method to return the initial amount converted to yen.

```
a.Balance

ans =

      1.4493e+04
```

MATLAB also calls get methods when it displays the object. When you set `Currency` to euros without ending the statement with a semicolon, MATLAB calls the `Balance` get method to display the updated value.

```
a.Currency = "EUR"

a =

  Account with properties:

         Currency: "EUR"
    USDollarAmount: 100
           Balance: 102.0400
```

When to Use Set Methods with Dependent Properties

Although dependent properties do not store values, you can still define set methods for them. The set methods cannot set the value of a dependent property, but they can execute other code.

For example, `propertyChange` is a value class that initially defined a property named `OldPropName`. You can use a set method to change the property name from the perspective of a class user:

- Redefine `OldPropName` as dependent and hidden.
- Define a new property with the name you would like to replace `OldPropName`.
- Define a set method for `OldPropName` that stores the value in `NewPropName`.
- Define a get method for `OldPropName` that returns the value stored in `NewPropName`.

```
classdef propertyChange
    properties
        NewPropName
    end
    properties (Dependent,Hidden)
        OldPropName
    end

    methods
        function obj = set.OldPropName(obj,val)
            obj.NewPropName = val;
        end
        function value = get.OldPropName(obj)
            value = obj.NewPropName;
        end
```

```
    end
end
```

Code that accesses `OldPropName` continues to work as expected, and making `OldPropName` hidden helps prevent new users from seeing the old property name.

For example, create an instance of `propertyChange`. Set the property value using the old property name and then display the object. MATLAB sets the value to the property with the new name and displays it.

```
a = propertyChange;
a.OldPropName = "hello"
```

```
p =

  propertyChange with properties:

    NewPropName: "hello"
```

See Also

Related Examples

- “Property Attributes” on page 8-8
- “Calculate Data on Demand” on page 3-17

Properties Containing Objects

In this section...

Assigning Objects as Default Property Values

MATLAB evaluates property default values only once when loading the class. MATLAB does not reevaluate the assignment each time you create an object of that class. If you assign an object as a default property value in the class definition, MATLAB calls the constructor for that object only once when loading the class.

Note Evaluation of property default values occurs only when the value is first needed, and only once when MATLAB first initializes the class. MATLAB does not reevaluate the expression each time you create an instance of the class.

For more information on the evaluation of expressions that you assign as property default values, see “When MATLAB Evaluates Expressions” on page 6-11.

Assigning to Read-Only Properties Containing Objects

When a class defines a property with private or protected `SetAccess`, and that property contains an object which itself has properties, assignment behavior depends on whether the property contains a handle or a value object:

- Handle object – you can set properties on handle objects contained in read-only properties
- Value object – you cannot set properties on value object contained in read-only properties.

Assignment Behavior

These classes illustrate the assignment behavior:

- `ReadOnlyProps` – class with two read-only properties. The class constructor assigns a handle object of type `HanClass` to the `PropHandle` property and a value object of type `ValClass` to the `PropValue` property.
- `HanClass` – handle class with public property
- `ValClass` – value class with public property

```
classdef ReadOnlyProps
   properties(SetAccess = private)
      PropHandle
      PropValue
   end
   methods
      function obj = ReadOnlyProps
         obj.PropHandle = HanClass;
```

```
         obj.PropValue = ValClass;
      end
   end
end

classdef HanClass < handle
   properties
      Hprop
   end
end

classdef ValClass
   properties
      Vprop
   end
end
```

Create an instance of the `ReadOnlyProps` class:

```
a = ReadOnlyProps
```

```
a =

 ReadOnlyProps with properties:

    PropHandle: [1x1 HanClass]
     PropValue: [1x1 ValClass]
```

Use the private `PropHandle` property to set the property of the `HanClass` object it contains:

```
class(a.PropHandle.Hprop)
```

```
ans =

double
```

```
a.PropHandle.Hprop = 7;
```

Attempting to make an assignment to the value class object property is not allowed:

```
a.PropValue.Vprop = 11;
```

```
You cannot set the read-only property 'PropValue' of ReadOnlyProps.
```

See Also

More About

- “Mutable and Immutable Properties” on page 8-16

Dynamic Properties — Adding Properties to an Instance

In this section...

What Are Dynamic Properties

You can add properties to instances of classes that derive from the `dynamicprops` class. These dynamic properties are sometimes referred to as instance properties. Use dynamic properties to attach temporary data to objects or to assign data that you want to associate with an instance of a class, but not all objects of that class.

It is possible for more than one program to define dynamic properties on the same object. In these cases, avoid name conflicts. Dynamic property names must be valid MATLAB identifiers (see “Variable Names”) and cannot be the same name as a method of the class.

Characteristics of Dynamic Properties

Once defined, dynamic properties behave much like class-defined properties:

- Set and query the values of dynamic properties using dot notation. (See “Assign Data to the Dynamic Property” on page 8-48.)
- MATLAB saves and loads dynamic properties when you save and load the objects to which they are attached. (See “Dynamic Properties and ConstructOnLoad” on page 8-57.)
- Define attributes for dynamic property. (See “Set Dynamic Property Attributes” on page 8-48.)
- By default, dynamic properties have their `NonCopyable` attribute set to `true`. If you copy an object containing a dynamic property, the dynamic property is not copied. (See “Objects with Dynamic Properties” on page 7-34.)
- Add property set and get access methods. (See “Set and Get Methods for Dynamic Properties” on page 8-51.)
- Listen for dynamic property events. (See “Dynamic Property Events” on page 8-53.)
- Access dynamic property values from object arrays, with restricted syntax. (See “Accessing Dynamic Properties in Arrays” on page 10-11.)
- The `isequal` function always returns `false` when comparing objects that have dynamic properties, even if the properties have the same name and value. To compare objects that contain dynamic properties, overload `isequal` for your class.

Define Dynamic Properties

Any class that is a subclass of the `dynamicprops` class (which is itself a subclass of the `handle` class) can define dynamic properties using the `addprop` method. The syntax is:

```
P = addprop(H,'PropertyName')
```

where:

`P` is an array of `meta.DynamicProperty` objects

`H` is an array of handles

`PropertyName` is the name of the dynamic property you are adding to each object

Naming Dynamic Properties

Use only valid names when naming dynamic properties (see “Variable Names”). In addition, *do not* use names that:

- Are the same as the name of a class method
- Are the same as the name of a class event
- Contain a period (`.`)
- Are the names of function that support array functionality: `empty`, `transpose`, `ctranspose`, `permute`, `reshape`, `display`, `disp`, `details`, or `sort`.

Set Dynamic Property Attributes

To set property attributes, use the `meta.DynamicProperty` object associated with the dynamic property. For example, if `P` is the object returned by `addprop`, this statement sets the property’s `Hidden` attribute to `true`:

```
P.Hidden = true;
```

The property attributes `Constant` and `Abstract` have no meaning for dynamic properties. Setting the value of these attributes to `true` has no effect.

Remove a Dynamic Property

Remove the dynamic property by deleting its `meta.DynamicProperty` object:

```
delete(P);
```

Assign Data to the Dynamic Property

Suppose, you are using a predefined set of user interface widget classes (buttons, sliders, check boxes, etc.). You want to store the location of each instance of the widget class. Assume that the widget classes are not designed to store location data for your particular layout scheme. You want to avoid creating a map or hash table to maintain this information separately.

Assuming the `button` class is a subclass of `dynamicprops`, add a dynamic property to store your layout data. Here is a simple class to create a `uicontrol` button:

```
classdef button < dynamicprops
   properties
      UiHandle
   end
   methods
      function obj = button(pos)
         if nargin > 0
            if length(pos) == 4
               obj.UiHandle = uicontrol('Position',pos,...
                  'Style','pushbutton');
            else
               error('Improper position')
            end
         end
```

```
      end
   end
end
```

Create an instance of the `button` class, add a dynamic property, and set its value:

```
b1 = button([20 40 80 20]);
b1.addprop('myCoord');
b1.myCoord = [2,3];
```

Access the dynamic property just like any other property, but only on the object on which you defined it:

```
b1.myCoord
```

```
ans =

     2     3
```

Access Attribute for Dynamic Properties

Using nonpublic Access with dynamic properties is not recommended because these properties belong to specific instances that are often created outside of class methods. The Access attribute of a dynamic property applies to the class of the instance that contains the dynamic property. The dynamic property Access attribute does not necessarily apply to the class whose method adds the dynamic property.

For example, if a base class method adds a dynamic property with private access to an instance, the private access applies only to the class of the instance.

For more information on dynamic property attributes, see `meta.DynamicProperty`. Use the handle `findprop` method to get the `meta.DynamicProperty` object.

List Object Dynamic Properties

You can list the dynamic properties for an object using the handle `findprop` method. Here are the steps:

- Get the names of the object's properties using the `properties` function.
- Get the metadata object for each property using `findprop`.
- Use the `isa` function to determine if the metadata object is a `meta.DynamicProperty` object. If so, then the property is a dynamic property.

The `getDynamicPropNames` function shows how to display the names of any dynamic properties defined for the input `obj`.

```
function getDynamicPropNames(obj)
    % Find dynamic properties
    allprops = properties(obj);
    for i=1:numel(allprops)
        m = findprop(obj,allprops{i});
        if isa(m,'meta.DynamicProperty')
            disp(m.Name)
        end
    end
end
```

See Also

Related Examples

- “Set and Get Methods for Dynamic Properties” on page 8-51
- “Dynamic Property Events” on page 8-53
- “Dynamic Properties and ConstructOnLoad” on page 8-57

Set and Get Methods for Dynamic Properties

You can define property set access or get access methods for dynamic properties without creating additional class methods. For general information on the use of access methods, see “Property Get and Set Methods” on page 8-38.

Create Access Methods for Dynamic Properties

Use these steps to create a property access method:

- Define a function that implements the operations you want to perform before the property set or get occurs. These methods must have the following signatures: `mySet(obj,val)` or `val = myGet(obj)`
- Obtain the dynamic property's corresponding `meta.DynamicProperty` object.
- Assign a function handle referencing your set or get property function to the `meta.DynamicProperty` object's `GetMethod` or `SetMethod` property. This function does not need to be a method of the class. You cannot use a naming scheme like `set.`*`PropertyName`*. Instead, use any other valid function name.

Suppose that you want to create a property set function for the `myCoord` dynamic property of the `button` class created in “Define Dynamic Properties” on page 8-47.

Write the function as follows.

```
function set_myCoord(obj,val)
   if  ~(length(val) == 2)
      error('myCoords require two values')
   end
   obj.myCoord = val;
end
```

Because `button` is a `handle` class, the property set function does not need to return the object as an output argument.

To get the `meta.DynamicProperty` object, use the `handle` class `findprop` method:

```
mb1 = b1.findprop('myCoord');
mb1.SetMethod = @set_myCoord;
```

MATLAB calls the property set function whenever you set this property:

```
b1.myCoord = [1 2 3] % length must be two

Error using button.set_myCoord
myCoords require two values
```

You can set and get the property values only from within your property access methods. You cannot call another function from the set or get method, and then attempt to access the property value from that function.

See Also

Related Examples

- “Dynamic Properties — Adding Properties to an Instance” on page 8-47

Dynamic Property Events

In this section...

Dynamic Properties and Ordinary Property Events

Dynamic properties support property set and get events so you can define listeners for these properties. Listeners are bound to the particular dynamic property for which they are defined.

If you delete a dynamic property, and then create another dynamic property with the same name, the listeners do not respond to events generated by the new property. A listener defined for a dynamic property that has been deleted does not cause an error, but the listener callback is never executed.

“Property-Set and Query Events” on page 11-14 provides more information on how to define listeners for these events.

Dynamic-Property Events

To respond to the addition and removal of dynamic properties, attach listeners to objects containing the dynamic properties. The `dynamicprops` class defines events for this purpose:

- `PropertyAdded` — Triggered when you add a dynamic property to an object derived from the `dynamicprops` class.
- `PropertyRemoved` — Triggered when you delete the object or the `meta.DynamicProperty` object associated with a dynamic property.
- `ObjectBeingDestroyed` — Triggered when the object is destroyed. This event is inherited from the `handle` class.

These events have public listen access (`ListenAccess` attribute) and private notify access (`NotifyAccess` attribute).

The `PropertyAdded` and `PropertyRemoved` events pass an `event.DynamicPropertyEvent` object to listener callbacks. The event data object has three properties:

- `PropertyName` — Name of the dynamic property that is added or removed
- `Source` — Handle to the object that is the source of the event
- `EventName` — Name of the event (`PropertyAdded`, `PropertyRemoved`, or `ObjectBeingDestroyed`)

Listen for a Specific Property Name

Suppose that you have an application that creates a dynamic property under certain conditions. You want to:

- Set the value of a hidden property to `true` when a property named `SpecialProp` is added.
- Set the value of the hidden property to `false` when `SpecialProp` is removed.

Use the `event.DynamicPropertyEvent` event data to determine the name of the property and whether it is added or deleted.

The `DynamTest` class derives from `dynamicprops`. It defines a hidden property, `HiddenProp`.

```
classdef DynamTest < dynamicprops
   properties (Hidden)
      HiddenProp
   end
end
```

Define a callback function that uses the `EventName` property of the event data to determine if a property is added or removed. Obtain the name of the property from the `PropertyName` property of the event data. If a dynamic property is named `SpecialProp`, change the value of the hidden property.

```
function DyPropEvtCb(src,evt)
   switch evt.EventName
      case 'PropertyAdded'
         switch evt.PropertyName
            case 'SpecialProp'
               % Take action based on the addition of this property
               %...
               %...
               src.HiddenProp = true;
               disp('SpecialProp added')
            otherwise
               % Other property added
               % ...
               disp([evt.PropertyName,' added'])
         end
      case 'PropertyRemoved'
         switch evt.PropertyName
            case 'SpecialProp'
               % Take action based on the removal of this property
               %...
               %...
               src.HiddenProp = false;
               disp('SpecialProp removed')
            otherwise
               % Other property removed
               % ...
               disp([evt.PropertyName,' removed'])
         end
   end
end
```

Create an object of the `DynamTest` class.

```
dt = DynamTest;
```

Add a listener for both `PropertyAdded` and `PropertyRemoved` events.

```
lad = addlistener(dt,'PropertyAdded',@DyPropEvtCb);
lrm = addlistener(dt,'PropertyRemoved',@DyPropEvtCb);
```

PropertyAdded Event Callback Execution

Adding a dynamic property triggers the `PropertyAdded` event. This statement adds a dynamic property to the object and saves the returned `meta.DynamicProperty` object.

```
ad = addprop(dt,'SpecialProp');
```

The addition of the dynamic property causes the listener to execute its callback function, `DyPropEvtCb`. The callback function assigns a value of `true` to the `HiddenProp` property.

```
dt.HiddenProp
```

```
ans =

     1
```

PropertyRemoved Event Callback Execution

Remove a dynamic property by calling `delete` on the `meta.DynamicProperty` object that is returned by the `addprop` method. Removing the `meta.DynamicProperty` object triggers the `PropertyRemoved` event.

Delete the `meta.DynamicProperty` object returned when adding the dynamic property `SpecialProp`.

```
delete(ad)
```

The callback executes:

```
SpecialProp removed
```

The value of `HiddenProp` is now `false`.

```
dt.HiddenProp
```

```
ans =

     0
```

How to Find meta.DynamicProperty Objects

You can obtain the `meta.DynamicProperty` object for a dynamic property using `findprop`. Use `findprop` if you do not have the object returned by `addprop`.

```
ad = findprop(dt,'SpecialProp');
```

See Also

Related Examples

- “Dynamic Properties — Adding Properties to an Instance” on page 8-47

Dynamic Properties and ConstructOnLoad

Setting the class `ConstructOnLoad` attribute to `true` causes MATLAB to call the class constructor when loading the class. MATLAB saves and restores dynamic properties when loading an object.

If you create dynamic properties from the class constructor, you can cause a conflict if you also set the class `ConstructOnLoad` attribute to `true`. Here is the sequence:

- A saved object saves the names and values of properties, including dynamic properties
- When loaded, a new object is created and all properties are restored to the values at the time the object was saved
- Then, the `ConstructOnLoad` attribute causes a call to the class constructor, which would create another dynamic property with the same name as the loaded property. See “Save and Load Objects” on page 13-2 for more on the load sequence.
- MATLAB prevents a conflict by loading the saved dynamic property, and does not execute `addprop` when calling the constructor.

If you use `ConstructOnLoad`, add dynamic properties from the class constructor, and want the constructor to call `addprop` at load time, then set the dynamic property `Transient` attribute to `true`. This setting prevents the property from being saved. For example:

```
classdef (ConstructOnLoad) MyClass < dynamicprops
   function obj = MyClass
      P = addprop(obj,'DynProp');
      P.Transient = true;
      ...
   end
end
```

See Also

Related Examples

- “Dynamic Properties — Adding Properties to an Instance” on page 8-47

Methods — Defining Class Operations

Methods in Class Design

In this section...

Class Methods

Methods are functions that implement the operations performed on objects of a class. Methods, along with other class members support the concept of encapsulation—class instances contain data in properties and class methods operate on that data. This design allows the internal workings of classes to be hidden from code outside of the class, and thereby enabling the class implementation to change without affecting code that is external to the class.

Methods have access to private members of their class including other methods and properties. This encapsulation enables you to hide data and create special interfaces that must be used to access the data stored in objects.

Examples and Syntax

For an example to get started writing classes, see “Creating a Simple Class” on page 2-2.

For sample code and syntax, see “Method Syntax” on page 5-7.

For a discussion of how to create classes that modify standard MATLAB behavior, see “Methods That Modify Default Behavior” on page 17-2 .

For information on the use of @ and path directors and packages to organize your class files, see “Class File Organization”.

For the syntax to use when defining classes in more than one file, see “Methods in Separate Files” on page 9-8.

Kinds of Methods

There are specialized kinds of methods that perform certain functions or behave in particular ways:

- *Ordinary methods* are functions that act on one or more objects and return some new object or some computed value. These methods are like ordinary MATLAB functions that cannot modify input arguments. Ordinary methods enable classes to implement arithmetic operators and computational functions. These methods require an object of the class on which to operate. See “Ordinary Methods” on page 9-6.
- *Constructor methods* are specialized methods that create objects of the class. A constructor method must have the same name as the class and typically initializes property values with data obtained from input arguments. The class constructor method must declare at least one output argument, which is the object being constructed. The first output is always the object being constructed. See “Class Constructor Methods” on page 9-15.

- *Destructor methods* are called automatically when the object is destroyed, for example if you call `delete(object)` or there are no longer any references to the object. See “Handle Class Destructor” on page 7-13.
- *Property access methods* enable a class to define code to execute whenever a property value is queried or set. See “Property Get and Set Methods” on page 8-38.
- *Static methods* are functions that are associated with a class, but do not necessarily operate on class objects. These methods do not require an instance of the class to be referenced during invocation of the method, but typically perform operations in a way specific to the class. See “Static Methods” on page 9-23.
- *Conversion methods* are overloaded constructor methods from other classes that enable your class to convert its own objects to the class of the overloaded constructor. For example, if your class implements a `double` method, then this method is called instead of the double class constructor to convert your class object to a MATLAB double object. See “Object Converters” on page 17-5 for more information.
- *Abstract methods* define a class that cannot be instantiated itself, but serves as a way to define a common interface used by numerous subclasses. Classes that contain abstract methods are often referred to as interfaces. See “Abstract Classes and Class Members” on page 12-68 for more information and examples.

Method Naming

The name of a function that implements a method can contain dots (for example, `set.PropertyName`) only if the method is one of the following:

- Property set/get access method (see “Property Get and Set Methods” on page 8-38)
- Conversion method that converts to a package-qualified class, which requires the use of the package name (see “Packages Create Namespaces” on page 6-21)

You cannot define property access or conversion methods as local functions, nested functions, or separately in their own files. Class constructors and package-scoped functions must use the unqualified name in the function definition; do not include the package name in the function definition statement.

See Also

Related Examples

- “Method Attributes” on page 9-4
- “Rules for Naming to Avoid Conflicts” on page 9-27

Method Attributes

In this section...

Purpose of Method Attributes

Specifying attributes in the class definition enables you to customize the behavior of methods for specific purposes. Control characteristics like access, visibility, and implementation by setting method attributes. Subclasses do not inherit superclass member attributes.

Specifying Method Attributes

Assign method attributes on the same line as the `methods` keyword:

```
methods (Attribute1 = value1, Attribute2 = value2,...)
   ...
end
```

Table of Method Attributes

Attributes enable you to modify the behavior of methods. All methods support the attributes listed in the following table.

Attribute values apply to all methods defined within the `methods...end` code block that specifies the nondefault values.

Method Attributes

Attribute Name	Class	Description
`Abstract`	`logical` Default = `false`	If `true`, the method has no implementation. The method has a syntax line that can include arguments that subclasses use when implementing the method: • Subclasses are not required to define the same number of input and output arguments. However, subclasses generally use the same signature when implementing their version of the method. • The method does not contain `function` or `end` keywords, only the function syntax (for example, `[a,b] = myMethod(x,y)`). • The method can include comments after the signature line.

`Access`	• enumeration, default = `public`
	• `meta.class` object
	• cell array of `meta.class` objects
		Determines what code can call this method:
		• `public` — Unrestricted access
		• `protected` — Access from methods in class or subclasses
		• `private` — Access by class methods only (not from subclasses)
		• List classes that have access to this method. Specify classes as `meta.class` objects in the form:
		  • A single `meta.class` object
		  • A cell array of `meta.class` objects. An empty cell array, `{}`, is the same as `private` access.
		See “Class Members Access” on page 12-23

`Hidden`	`logical` Default = `false`	When `false`, the method name shows in the list of methods displayed using the `methods` or `methodsview` commands. If set to `true`, the method name is not included in these listings and `ismethod` does not return `true` for this method name.
`Sealed`	`logical` Default = `false`	If `true`, the method cannot be redefined in a subclass. Attempting to define a method with the same name in a subclass causes an error.
`Static`	`logical` Default = `false`	Specify as `true` to define a method that does not depend on an object of the class and does not require an object argument. Use the class name to call the method: *`classname.methodname`* or an instance of the class: `obj.`*`methodname`* “Static Methods” on page 9-23 provides more information.
Framework attributes	Classes that use certain framework base classes have framework-specific attributes. See the documentation for the specific base class you are using for information on these attributes.	

See Also

`metaclass` | `meta.method`

More About

- “Methods”

Ordinary Methods

In this section...

Ordinary Methods Operate on Objects

Ordinary methods define functions that operate on objects of the class. Therefore, one of the input arguments must be an object or array of objects of the defining class. These methods can compute values based on object data, can overload MATLAB built-in functions, and can call other methods and functions. Ordinary methods can return modified objects.

Methods Inside classdef Block

This example shows the definition of a method (*`methodName`*) within the `classdef` and `methods` blocks:

```
classdef ClassName
   methods (AttributeName = value,...)
      function methodName(obj,args)
         % method code
         ...
      end
      ...
   end % end of method block
   ...
end
```

Method attributes apply only to that particular methods block, which is terminated by the `end` statement.

Note Nonstatic methods must include an explicit object variable as a function argument. The MATLAB language does not support an implicit reference in the method function definition.

Example of a Method

The `addData` method adds a value to the `Data` property of `MyData` objects. The `mustBeNumeric` function restricts the value of the `Data` property to numeric values. The property has a default value of `0`.

The `addData` method returns the modified object, which you can reassign to the same variable.

```
classdef MyData
   properties
      Data {mustBeNumeric} = 0
   end
   methods
      function obj = addData(obj,val)
         if isnumeric(val)
```

```
            newData = obj.Data + val;
            obj.Data = newData;
        end
    end
  end
end

a = MyData;
a = addData(a,75)

a =

  MyData with properties:

    Data: 75
```

Method Files

You can define methods:

- Inside the class definition block
- In a separate file in the class folder (that is, `@ClassName` folder)

For more information on class folders, see “Folders Containing Class Definitions” on page 6-14.

See Also

More About

- “Methods in Separate Files” on page 9-8
- “Method Invocation” on page 9-11
- “Operator Overloading” on page 17-19

Methods in Separate Files

In this section...

Class Folders

You can define class methods in files that are separate from the class definition file, with certain exceptions (see “Methods You Must Define in the classdef File” on page 9-10).

To use multiple files for class definitions, put the class files in a folder having a name beginning with the `@` character followed by the name of the class (this is called a class folder). Ensure that the parent folder of the class folder is on the MATLAB path.

If the class folder is contained in one or more package folders, then the top-level package folder must be on the MATLAB path.

For example, the folder `@MyClass` must contain the file `MyClass.m` (which contains the `classdef` block) and contains other methods and function defined in files having a `.m` extension. The folder `@MyClass` can contain a number of files:

```
@MyClass/MyClass.m
@MyClass/subsref.m
@MyClass/subsasgn.m
@MyClass/horzcat.m
@MyClass/vertcat.m
@MyClass/myFunc.m
```

Types of Method Files

MATLAB treats any function file in the class folder as a method of the class. Function files can be MATLAB code (`.m`), Live Code file format (`.mlx`), MEX functions (platform dependent extensions), and P-code files (`.p`). The base name of the file must be a valid MATLAB function name. Valid function names begin with an alphabetic character, and can contain letters, numbers, or underscores.

For information on defining methods as C++ MEX functions, see “Using MEX Functions for MATLAB Class Methods”.

Define Method in Function File

To define a method in a separate file in the class folder, create the function in a file. Do not use the `method-end` keywords in that file. Name the file with the function name, as with any function.

In the `myFunc.m` file, implement the method:

```
function output = myFunc(obj,arg1,arg2)
   ...% code here
end
```

It is a good practice to declare the function signature in the `classdef` file in a methods block:

```
classdef MyClass
   methods
      output = myFunc(obj,arg1,arg2)
   end
   ...
end
```

Specify Method Attributes in classdef File

If you specify method attributes for a method that you define in a separate function file, include the method signature in a `methods` block in the `classdef` file. This methods block specifies the attributes that apply to the method.

For example, the following code shows a method with `Access` set to `private` in the `methods` block. The method implementation resides in a separate file. Do not include the `function` or `end` keywords in the `methods` block. Include only the function signature showing input and output arguments.

```
classdef MyClass
   methods (Access = private)
      output = myFunc(obj,arg1,arg2)
   end
end
```

In a file named `myFunc.m`, in the `@MyClass` folder, define the function:

```
function output = myFunc(obj,arg1,arg2)
   ...
end
```

Static Methods in Separate Files

To create a static method, set the method `Static` attribute to `true` and list the function signature in a static methods block in the `classdef` file. Include the input and output arguments with the function name. For example:

```
classdef MyClass
...
   methods (Static)
      output = staticFunc1(arg1,arg2)
      staticFunc2
   end
   ...
end
```

Define the functions in separate files using the same function signature. For example, in the file `@MyClass/staticFunc1.m`:

```
function output = staticFunc1(arg1,arg2)
   ...
end
```

and in `@Myclass/staticFunc2.m`:

```
function staticFunc2
   ...
end
```

Methods You Must Define in the classdef File

Define the following methods in the `classdef` file. You cannot define these methods in separate files:

- Class constructor
- All functions that use dots in their names, including:
  - Converter methods that must use the package name as part of the class name because the class is contained in packages
  - Property set and get access methods

Related Information

- “Converters for Package Classes” on page 17-5
- “Property Get and Set Methods” on page 8-38

See Also

Related Examples

- “Folders Containing Class Definitions” on page 6-14
- “Live Code File Format (.mlx)”
- “Call MEX Functions”
- “Using MEX Functions for MATLAB Class Methods”
- “Security Considerations to Protect Your Source Code”

Method Invocation

In this section...

MATLAB classes support both dot and function syntaxes for invoking methods. This topic demonstrates both syntaxes and describes how MATLAB determines what method to invoke.

Dot and Function Syntaxes

To invoke a nonstatic method with one argument `arg`, where `obj` is an object of the class that defines the method, use dot syntax or function syntax.

```
obj.methodName(arg)
methodName(obj,arg)
```

For example, `dataSetSummary` stores a set of numeric data along with the mean, median, and range of that data. The class defines two methods: `showDataSet` displays the current data stored in the data property, and `newDataSet` replaces the current value of `data` and calculates the mean, median, and range of that data.

```
classdef dataSetSummary
    properties (SetAccess=private)
        data {mustBeNumeric}
        dataMean
        dataMedian
        dataRange
    end

    methods
        function showDataSet(obj)
            disp(obj.data)
        end
        function obj = newDataSet(obj,inputData)
            obj.data = inputData;
            obj.dataMean = mean(inputData);
            obj.dataMedian = median(inputData);
            obj.dataRange = range(inputData);
        end
    end
end
```

Create an instance of `dataSetSummary` and invoke `newDataSet` to add data to the object. Use dot syntax to call `newDataSet`. Because `dataSetSummary` is a value class, assign the result back to the original variable to preserve the change.

```
a = dataSetSummary;
a = a.newDataSet([1 2 3 4])

a =

  dataSetSummary with properties:

          data: [1 2 3 4]
```

```
      dataMean: 2.5000
    dataMedian: 2.5000
     dataRange: 3
```

Invoke the `showDataSet` method, but use function syntax for this call.

```
showDataSet(a)
```

```
     1     2     3     4
```

Referencing Method Names with Expressions

You can invoke a class method dynamically by enclosing an expression in parentheses.

```
obj.(expression)
```

The expression must evaluate to a character vector or string that is the name of a method. For example, these two statements are equivalent for an object `a` of class `dataSetSummary`.

```
a.showDataSet
a.("showDataSet")
```

This technique does not work when used with function syntax.

Indexing into the Result of Method Call

You can dot index into the result of any method that returns a value for which dot indexing is defined, such as an object property or structure field name. For example, add a new method `returnSummary` to the `dataSetSummary` class that returns all of the stored data in a struct.

```
function outStruct = returnSummary(obj)
   outStruct = struct("Data",obj.data,...
                      "Mean",obj.dataMean,...
                      "Median",obj.dataMedian,...
                      "Range",obj.dataRange);
end
```

Call `returnSummary` and use dot indexing to return the median of the data set.

```
a.returnSummary.Median
```

```
ans =

    2.5000
```

For more information on indexing into the result of function calls, see “Indexing into Function Call Results”.

Determining Which Method Is Invoked

When dot syntax is used to invoke a method, MATLAB calls the method defined by the class of the object to the left of the dot. For example, if `classA` and `classB` both define a method called `plus`, then this code always invokes the `plus` method defined by `classA`.

```
A = classA;
B = classB;
A.plus(B)
```

No other arguments are considered. Methods of other arguments are never called, nor are functions.

In other syntaxes, MATLAB must determine which of possibly many versions of an operator or function to call in a given situation. The default behavior is to call the method associated with the leftmost argument. In both of these statements, the `plus` method defined by `classA` is called.

```
objA + objB
plus(objA,objB)
```

However, this default behavior can be changed when one object has precedence over another.

Object Precedence

Depending on how classes are defined, the objects of those classes can take precedence over other objects when it comes to method dispatching:

- Classes defined with the `classdef` syntax have precedence over these MATLAB classes:

  `double`, `single`, `int64`, `uint64`, `int32`, `uint32`, `int16`, `uint16`, `int8`, `uint8`, `char`, `string`, `logical`, `cell`, `struct`, and `function_handle`.
- Classes defined with the `classdef` syntax can specify their relative precedence to other classes using the `InferiorClasses` attribute.

In “Representing Polynomials with Classes” on page 19-2, the `DocPolynom` class defines a `plus` method that enables the addition of `DocPolynom` objects. Construct a `DocPolynom` instance.

```
p = DocPolynom([1 0 -2 -5])

p =
    x^3 - 2*x - 5
```

This statement adds a double to the `DocPolynom` instance. The `DocPolynom` class is dominant over the built-in `double` class, even though the `double` is the leftmost argument in `1 + p`. This code invokes the `DocPolynom` `plus` method to add the polynomials.

```
1 + p

ans =
    x^3 - 2*x - 4
```

You can also specify the relative precedence of classes defined with the `classdef` syntax by listing inferior classes in a class attribute. The `InferiorClasses` attribute gives the class higher precedence than the classes listed as arguments for the attribute. Define the `InferiorClasses` attribute in the `classdef` statement:

```
classdef (InferiorClasses = {?class1,?class2}) myClass
```

This attribute establishes a relative priority of the class being defined with the order of the classes listed. For more information, see “Class Precedence” on page 6-19.

See Also

More About

- “Class Attributes” on page 6-5

- “Method Attributes” on page 9-4
- “Class Precedence” on page 6-19
- “Call Superclass Methods on Subclass Objects” on page 5-11

Class Constructor Methods

In this section...

Purpose of Class Constructor Methods

A constructor method is a special function that creates an instance of the class. Typically, constructor methods accept input arguments to assign the data stored in properties and return an initialized object.

For a basic example, see “Creating a Simple Class” on page 2-2.

MATLAB classes that do not explicitly define any class constructors have a default constructor method. This method returns an object of the class that is created with no input arguments. A class can define a constructor method that overrides the default constructor. An explicitly defined constructor can accept input arguments, initialize property values, call other methods, and perform other operations necessary to create objects of the class.

Basic Structure of Constructor Methods

Constructor methods can be structured into three basic sections:

- Pre-initialization — Compute arguments for superclass constructors.
- Object initialization — Call superclass constructors.
- Post initialization — Perform any operations related to the subclass, including referencing and assigning to the object, call class methods, passing the object to functions, and so on.

This code illustrates the basic operations performed in each section:

```
classdef ConstructorDesign < BaseClass1
   properties
      ComputedValue
   end
   methods
```

```
        function obj = ConstructorDesign(a,b,c)

           %% Pre Initialization %%
           % Any code not using output argument (obj)
           if nargin == 0
              % Provide values for superclass constructor
              % and initialize other inputs
              a = someDefaultValue;
              args{1} = someDefaultValue;
              args{2} = someDefaultValue;
           else
              % When nargin ~= 0, assign to cell array,
              % which is passed to supclass constructor
              args{1} = b;
              args{2} = c;
           end
           compvalue = myClass.staticMethod(a);

           %% Object Initialization %%
           % Call superclass constructor before accessing object
           % You cannot conditionalize this statement
           obj = obj@BaseClass1(args{:});

           %% Post Initialization %%
           % Any code, including access to object
           obj.classMethod(arg);
           obj.ComputedValue = compvalue;
           ...
        end
     ...
     end
...
end
```

Call the constructor like any function, passing arguments and returning an object of the class.

```
obj = ConstructorDesign(a,b,c);
```

Guidelines for Constructors

- The constructor has the same name as the class.
- The constructor can return multiple arguments, but the first output must be the object created.
- If you do not want to assign the output argument, you can clear the object variable in the constructor (see “Output Object Suppressed” on page 9-22).
- If you create a class constructor, ensure it can be called with no input arguments. See “No Input Argument Constructor Requirement” on page 9-18.
- If your constructor makes an explicit call to a superclass constructor, this call must occur before any other reference to the constructed object and cannot occur after a `return` statement.
- Calls to superclass constructors cannot be conditional. You cannot place superclass construction calls in loops, conditions, switches, try/catch, or nested functions. See “No Conditional Calls to Superclass Constructors” on page 9-19 for more information.

Default Constructor

If a class does not define a constructor, MATLAB supplies a default constructor that takes no arguments and returns a scalar object whose properties are initialized to property default values. The default constructor supplied by MATLAB also calls all superclass constructors with no arguments or with any argument passed to the default subclass constructor.

When a subclass does not define a constructor, the default constructor passes its inputs to the direct superclass constructor. This behavior is useful when there is no need for a subclass to define a constructor, but the superclass constructor does require input arguments.

When to Define Constructors

Define a constructor method to perform object initialization that a default constructor cannot perform. For example, when creating an object of the class requires:

- Input arguments
- Initializing object state, such as property values, for each instance of the class
- Calling the superclass constructor with values that are determined by the subclass constructor

Related Information

For information specific to constructing enumerations, see “Enumeration Class Constructor Calling Sequence” on page 14-7.

For information on creating object arrays in the constructor, see “Construct Object Arrays” on page 10-2.

If the class being created is a subclass, MATLAB calls the constructor of each superclass class to initialize the object. Implicit calls to the superclass constructor are made with no arguments. If superclass constructors require arguments, call them from the subclass constructor explicitly. See “Control Sequence of Constructor Calls” on page 12-11

Initializing Objects in Constructor

Constructor methods return an initialized object as an output argument. The output argument is created when the constructor executes, before executing the first line of code.

For example, the following constructor can assign the value of the object's property `A` as the first statement because the object `obj` has already been assigned to an instance of `MyClass`.

```
function obj = MyClass(a,b,c)
   obj.A = a;
   ...
end
```

You can call other class methods from the constructor because the object is already initialized.

The constructor also creates an object whose properties have their default values — either empty (`[]`) or the default value specified in the property definition block.

For example, this constructor operates on the input arguments to assign the value of the `Value` property.

```
function obj = MyClass(a,b,c)
   obj.Value = (a + b) / c;
   ...
end
```

Referencing the Object in a Constructor

When initializing the object, for example, by assigning values to properties, use the name of the output argument to refer to the object within the constructor. For example, in the following code the output argument is `obj` and the object is reference as `obj`:

```
% obj is the object being constructed
function obj = MyClass(arg)
   obj.propert1 = arg*10;
   obj.method1;
   ...
end
```

For more information on defining default property values, see “Define Properties with Default Values” on page 8-13.

No Input Argument Constructor Requirement

There are cases where the constructor must be able to be called with no input argument:

- When loading objects into the workspace, if the class `ConstructOnLoad` attribute is set to `true`, the `load` function calls the class constructor with no arguments.
- When creating or expanding an object array such that not all elements are given specific values, the class constructor is called with no arguments to fill in unspecified elements (for example, `x(10,1) = MyClass(a,b,c);`). In this case, the constructor is called once with no arguments to populate the empty array elements (`x(1:9,1)`) with copies of this one object.

If there are no input arguments, the constructor creates an object using only default properties values. A good practice is to add a check for zero arguments to the class constructor to prevent an error if either of these two cases occur:

```
function obj = MyClass(a,b,c)
   if  nargin > 0
      obj.A = a;
      obj.B = b;
      obj.C = c;
      ...
   end
end
```

For ways to handle superclass constructors, see “Basic Structure of Constructor Methods” on page 9-15.

Subclass Constructors

Subclass constructors can call superclass constructors explicitly to pass arguments to the superclass constructor. The subclass constructor must specify these arguments in the call to the superclass constructor and must use the constructor output argument to form the call. Here is the syntax:

```
classdef MyClass < SuperClass
   methods
      function obj = MyClass(a,b,c,d)
         obj@SuperClass(a,b);
         ...
      end
   end
end
```

The subclass constructor must make all calls to superclass constructors before any other references to the object (`obj`). This restriction includes assigning property values or calling ordinary class methods. Also, a subclass constructor can call a superclass constructor only once.

Reference Only Specified Superclasses

If the `classdef` does not specify the class as a superclass, the constructor cannot call a superclass constructor with this syntax. That is, subclass constructor can call only direct superclass constructors listed in the `classdef` line.

```
classdef MyClass < SuperClass1 & SuperClass2
```

MATLAB calls any uncalled constructors in the left-to-right order in which they are specified in the `classdef` line. MATLAB passes no arguments with these calls.

No Conditional Calls to Superclass Constructors

Calls to superclass constructors must be unconditional. There can be only one call for a given superclass. Initialize the superclass portion of the object by calling the superclass constructors before using the object (for example, to assign property values or call class methods).

To call a superclass constructor with different arguments that depend on some condition, build a cell array of arguments and provide one call to the constructor.

For example, the `Cube` class constructor calls the superclass `Shape` constructor using default values when the `Cube` constructor is called with no arguments. If the `Cube` constructor is called with four input arguments, then pass `upvector` and `viewangle` to the superclass constructor:

```
classdef Cube < Shape
   properties
      SideLength = 0
      Color = [0 0 0]
   end
   methods
      function cubeObj = Cube(length,color,upvector,viewangle)
         % Assemble superclass constructor arguments
         if nargin == 0
            super_args{1} = [0 0 1];
            super_args{2} = 10;
         elseif nargin == 4
            super_args{1} = upvector;
            super_args{2} = viewangle;
         else
            error('Wrong number of input arguments')
         end
```

```
            % Call superclass constructor
            cubeObj@Shape(super_args{:});

            % Assign property values if provided
            if nargin > 0
               cubeObj.SideLength = length;
               cubeObj.Color = color;
            end
            ...
         end
      ...
      end
end
```

Zero or More Superclass Arguments

To support a syntax that calls the superclass constructor with no arguments, provide this syntax explicitly.

Suppose in the case of the `Cube` class example, all property values in the `Shape` superclass and the `Cube` subclass have default values specified in the class definitions. Then you can create an instance of `Cube` without specifying any arguments for the superclass or subclass constructors.

Here is how you can implement this behavior in the `Cube` constructor:

```
methods
   function cubeObj = Cube(length,color,upvector,viewangle)
      % Assemble superclass constructor arguments
      if nargin == 0
         super_args = {};
      elseif nargin == 4
         super_args{1} = upvector;
         super_args{2} = viewangle;
      else
         error('Wrong number of input arguments')
      end

      % Call superclass constructor
      cubeObj@Shape(super_args{:});

      % Assign property values if provided
      if nargin > 0
         cubeObj.SideLength = length;
         cubeObj.Color = color;
      end
   ...
   end
end
```

More on Subclasses

See “Design Subclass Constructors” on page 12-7 for information on creating subclasses.

Implicit Call to Inherited Constructor

MATLAB passes arguments implicitly from a default subclass constructor to the superclass constructor. This behavior eliminates the need to implement a constructor method for a subclass only to pass arguments to the superclass constructor.

For example, the following class constructor requires one input argument (a `datetime` object), which the constructor assigns to the `CurrentDate` property.

```
classdef BaseClassWithConstr
   properties
      CurrentDate datetime
   end
   methods
      function obj = BaseClassWithConstr(dt)
         obj.CurrentDate = dt;
      end
   end
end
```

Suppose that you create a subclass of `BaseClassWithConstr`, but your subclass does not require an explicit constructor method.

```
classdef SubclassDefaultConstr < BaseClassWithConstr
   ...
end
```

You can construct an object of the `SubclassDefaultConstr` by calling its default constructor with the superclass argument:

```
obj = SubclassDefaultConstr(datetime);
```

For information on subclass constructors, see “Subclass Constructors” on page 9-18 and “Default Constructor” on page 9-17.

Errors During Class Construction

For handle classes, MATLAB calls the `delete` method when an error occurs under the following conditions:

- A reference to the object is present in the code prior to the error.
- An early `return` statement is present in the code before the error.

MATLAB calls the `delete` method on the object, the `delete` methods for any objects contained in properties, and the `delete` methods for any initialized base classes.

Depending on when the error occurs, MATLAB can call the class destructor before the object is fully constructed. Therefore class `delete` methods must be able to operate on partially constructed objects that might not have values for all properties. For more information, see “Support Destruction of Partially Constructed Objects” on page 7-15.

For information on how objects are destroyed, see “Handle Class Destructor” on page 7-13.

Output Object Suppressed

You can suppress the assignment of the class instance to the `ans` variable when no output variable is assigned in a call to the constructor. This technique is useful for apps that creates graphical interface windows that hold onto the constructed objects. These apps do not need to return the object.

Use `nargout` to determine if the constructor has been called with an output argument. For example, the class constructor for the `MyApp` class clears the object variable, `obj`, if called with no output assigned:

```
classdef MyApp
   methods
      function obj = MyApp
         ...
         if nargout == 0
            clear obj
         end
      end
      ...
   end
end
```

When a class constructor does not return an object, MATLAB does not trigger the `meta.class InstanceCreated` event.

See Also

Related Examples

- “Simplifying the Interface with a Constructor” on page 3-16
- “Subclass Constructor Implementation” on page 12-8

Static Methods

In this section...

What Are Static Methods

Static methods are associated with a class, but not with specific instances of that class. These methods do not require an object of the class as an input argument. Therefore, you can call static methods without creating an object of the class.

Why Define Static Methods

Static methods are useful when you do not want to create an instance of the class before executing some code. For example, suppose you want to set up the MATLAB environment or use the static method to calculate data required to create class instances.

Suppose that a class needs a value for pi calculated to particular tolerances. The class could define its own version of the built-in `pi` function for use within the class. This approach maintains the encapsulation of the class's internal workings, but does not require an instance of the class to return a value.

Defining Static Methods

To define a method as static, set the methods block `Static` attribute to `true`. For example:

```
classdef MyClass
   methods(Static)
       function p = pi(tol)
           [n d] = rat(pi,tol);
           p = n/d;
       end
   end
end
```

Calling Static Methods

Invoke static methods using the name of the class followed by dot (`.`), then the name of the method:

```
classname.staticMethodName(args,...)
```

Calling the `pi` method of `MyClass` in the previous section would require this statement:

```
value = MyClass.pi(.001);
```

You can also invoke static methods using an instance of the class, like any method:

```
obj = MyClass;
value = obj.pi(.001);
```

Inheriting Static Methods

Subclasses can redefine static methods unless the method's `Sealed` attribute is also set to `true` in the superclass.

See Also

Related Examples

- “Implementing the AccountManager Class” on page 3-11

Overload Functions in Class Definitions

In this section...

Why Overload Functions

Classes can redefine MATLAB functions by implementing methods having the same name. Overloading is useful when defining specialized types that you want to behave like existing MATLAB types. For example, you can implement relational operations, plotting functions, and other commonly used MATLAB functions to work with objects of your class.

You can also modify default behaviors by implementing specific functions that control these behaviors. For more information on functions that modify default behaviors, see “Methods That Modify Default Behavior” on page 17-2.

Implementing Overloaded MATLAB Functions

Class methods can provide implementations of MATLAB functions that operate only on instances of the class. This restriction is possible because MATLAB can always identify to which class an object belongs.

MATLAB uses the dominant argument to determine which version of a function to call. If the dominant argument is an object, then MATLAB calls the method defined by the object's class, if one exists.

In cases where a class defines a method with the same name as a global function, the class's implementation of the function is said to *overload* the original global implementation.

To overload a MATLAB function:

- Define a method with the same name as the function you want to overload.
- Ensure that the method argument list accepts an object of the class, which MATLAB uses to determine which version to call.
- Perform the necessary steps in the method to implement the function. For example, access the object properties to manipulate data.

Generally, the method that overloads a function produces results similar to the MATLAB function. However, there are no requirements regarding how you implement the overloading method. The overloading method does not need to match the signature of the overloaded function.

Note MATLAB does not support defining multiple methods with the same name but different signatures in the same class.

Overload the bar Function

It is convenient to overload commonly used functions to work with objects of your class. For example, suppose that a class defines a property that stores data that you often graph. The `MyData` class overrides the `bar` function and adds a title to the graph:

```
classdef MyData
   properties
      Data
   end
   methods
      function obj = MyData(d)
         if nargin > 0
            obj.Data = d;
         end
      end
      function bar(obj)
         y = obj.Data;
         bar(y,'EdgeColor','r');
         title('My Data Graph')
      end
   end
end
```

The `MyData` `bar` method has the same name as the MATLAB `bar` function. However, the `MyData` `bar` method requires a `MyData` object as input. Because the method is specialized for `MyData` objects, it can extract the data from the `Data` property and create a specialized graph.

To use the `bar` method, create an object:

```
y = rand(1,10);
md = MyData(y);
```

Call the method using the object:

```
bar(md)
```

You can also use dot notation:

```
md.bar
```

Implementing MATLAB Operators

Classes designed to implement new MATLAB data types typically define certain operators, such as addition, subtraction, or equality.

For example, standard MATLAB addition (`+`) cannot add two polynomials because this operation is not defined by simple addition. However, a `polynomial` class can define its own `plus` method that the MATLAB language calls to perform addition of `polynomial` objects when you use the `+` symbol:

```
p1 + p2
```

For information on overloading operators, see “Operator Overloading” on page 17-19.

Rules for Naming to Avoid Conflicts

The names of methods, properties, and events are scoped to the class. Therefore, adhere to the following rules to avoid naming conflicts:

- You can reuse names that you have used in unrelated classes.
- You can reuse names in subclasses if the member does not have public or protected access. These names then refer to entirely different methods, properties, and events without affecting the superclass definitions
- Within a class, all names exist in the same name space and must be unique. A class cannot define two methods with the same name and a class cannot define a local function with the same name as a method.
- The name of a static method is considered without its class prefix. Thus, a static method name without its class prefix cannot match the name of any other method.

See Also

Related Examples

- “Class Support for Array-Creation Functions” on page 9-28

Class Support for Array-Creation Functions

In this section...

Extend Array-Creation Functions for Your Class

There are several MATLAB functions that create arrays of a specific size and type, such as `ones` and `zeros`. User-defined classes can add support for array-creation functions without requiring the use of overloaded method syntax.

Class support for any of the array-creation functions enables you to develop code that you can share with built-in and user-defined data types. For example, the class of the variable `x` in the following code can be a built-in type during initial development, and then be replaced by a user-defined class that transparently overloads `zeros`:

```
cls = class(x);
zArray = zeros(m,n,cls);
```

Array-creation functions create arrays of a specific type in two ways:

- Class name syntax — Specify class name that determines the type of array elements.
- Prototype object syntax — Provide a prototype object that the function uses to determine the type and other characteristics of the array elements.

For example:

```
zArray = zeros(2,3,'uint8');
```

```
p = uint8([1 3 5; 2 4 6]);
zArray = zeros(2,3,'like',p);
```

After adding support for these functions to a class named `MyClass`, you can use similar syntax with that class:

```
zArray = zeros(2,3,'MyClass');
```

Or pass an object of your class:

```
p = MyClass(...);
zArray = zeros(size(p),'like',p);
```

MATLAB uses these arguments to dispatch to the appropriate method in your class.

Array-Creation Functions That Support Overloading

These functions support this kind of overloading:

Array-Creation Functions
`ones`

Array-Creation Functions
`zeros`
`eye`
`nan` (lowercase)
`inf`
`true`
`false`
`cast`
`rand`
`randn`
`randi`

Scalar Functions That Support Overloading

These functions also support similar overloading, with the exception that the output is always a scalar (or a 1-by-1 sparse matrix):

Scalar Functions
`eps`
`realmax`
`realmin`
`intmax`
`intmin`
`flintmax`

For these functions, you do not need to specify the size when creating scalars of a specific type. For example:

```
d = eps('single');
```

```
p = single([1 3 5; 2 4 6]);
d = eps('like',p);
```

After adding support for these functions to a user-defined class, you can use similar syntax with that class as well.

Which Syntax to Use

To create an array of default objects, which require no input arguments for the constructor, then use the class name syntax.

To create an array of objects with specific property values or if the constructor needs other inputs, use the prototype object to provide this information.

Classes can support both the class name and the prototype object syntax.

You can implement a class name syntax with the `true` and `false` functions even though these functions do not support that syntax by default.

Class Name Method Called If Prototype Method Does Not Exist

If your class implements a class name syntax, but does not implement a prototype object syntax for a particular function, you can still call both syntaxes. For example, if you implement a static `zeros` method only, you can call:

```
zeros(...,'like',MyClass(...))
```

In the case in which you call the prototype object syntax, MATLAB first searches for a method named `zerosLike`. If MATLAB cannot find this method, it calls for the `zeros` static method.

This feature is useful if you only need the class name to create the array. You do not need to implement both methods to support the complete array-creation function syntax. When you implement only the class name syntax, a call to a prototype object syntax is the same as the call to the class name syntax.

Implement Support for Array-Creation Functions

Use two separate methods to support an array-creation function. One method implements the class name syntax and the other implements the prototype object syntax.

For example, to support the `zeros` function:

- Implement the class name syntax:

  ```
  zeros(...,'ClassName')
  ```

  As a `Static` method:

  ```
  methods (Static)
     function z = zeros(varargin)
        ...
     end
  end
  ```

- Implement the prototype object syntax:

  ```
  zeros(...,'like',obj)
  ```

  As a `Hidden` method with the `char` vector `'Like'` appended to the name.

  ```
  methods (Hidden)
     function z = zerosLike(obj,varargin)
        ...
     end
  end
  ```

How MATLAB Interprets the Function Call

The special support for array-creation functions results from the interpretation of the syntax.

- A call to the `zeros` function of this form:

  ```
  zeros(...,'ClassName')
  ```

  Calls the class static method with this syntax:

  ```
  ClassName.zeros(varargin{1:end-1})
  ```

- A call to the `zeros` function of this form:

  ```
  zeros(...,'like',obj)
  ```

  Calls the class method with this syntax:

  ```
  zerosLike(obj,varargin{1:end-2})
  ```

Support All Function Inputs

The input arguments to an array-creation function can include the dimensions of the array the function returns and possibly other arguments. In general, there are three cases that your methods must support:

- No dimension input arguments resulting in the return of a scalar. For example:

  ```
  z = zeros('MyClass');
  ```

- One or more dimensions equal to or less than zero, resulting in an empty array. For example:

  ```
  z = zeros(2,0,'MyClass');
  ```

- Any number of valid array dimensions specifying the size of the array. For example:

  ```
  z = zeros(2,3,5,'MyClass');
  ```

When the array-creation function calls your class method, it passes the input arguments, excluding the class name or the literal `'like'` and the object variable to your method. You can implement your methods with these signatures:

- `zeros(varargin)` for “class name” methods
- `zeros(obj,varargin)` for “like prototype object” methods

Sample Class

The `Color` class represents a color in a specific color space, such as, `RGB`, `HSV`, and so on. The discussions in “Class Name Method Implementations” on page 9-31 and “Prototype Object Method Implementation” on page 9-33 use this class as a basis for the overloaded method implementations.

```
classdef Color
   properties
      ColorValues = [0,0,0]
      ColorSpace = 'RGB'
   end
   methods
      function obj = Color(cSpace,values)
         if nargin > 0
            obj.ColorSpace = cSpace;
            obj.ColorValues = values;
         end
      end
   end
end
```

Class Name Method Implementations

The `zeros` function strips the final *`ClassName`* `char` vector and uses it to form the call to the static method in the `Color` class. The arguments passed to the static method are the array dimension arguments.

Here is an implementation of a `zeros` method for the `Color` class. This implementation:

- Defines the `zeros` method as `Static` (required)
- Returns a scalar `Color` object if the call to `zeros` has no dimension arguments
- Returns an empty array if the call to `zeros` has any dimensions arguments equal to 0.
- Returns an array of default `Color` objects. Use `repmat` to create an array of the dimensions specified by the call to `zeros`.

```
classdef Color
   ...
   methods (Static)
      function z = zeros(varargin)
         if (nargin == 0)
         % For zeros('Color')
            z = Color;
         elseif any([varargin{:}] <= 0)
         % For zeros with any dimension <= 0
            z = Color.empty(varargin{:});
         else
         % For zeros(m,n,...,'Color')
         % Use property default values
            z = repmat(Color,varargin{:});
         end
      end
   end
end
```

The `zeros` method uses default values for the `ColorValues` property because these values are appropriate for this application. An implementation of a `ones` method can set the `ColorValues` property to `[1,1,1]`, for example.

Suppose that you want to overload the `randi` function to achieve the following objectives:

- Define each `ColorValue` property as a 1-by-3 array in the range of 1 to a specified maximum value (for example, 1–255).
- Accommodate scalar, empty, and multidimensional array sizes.
- Return an array of `Color` objects of the specified dimensions, each with random `ColorValues`.

```
classdef Color
   ...
   methods (Static)
      function r = randi(varargin)
         if (nargin == 0)
            % For randi('ClassName')
            r = Color('RGB',randi(255,[1,3]));
         elseif any([varargin{2:end}] <= 0)
            % For randi with any dimension <= 0
            r = Color.empty(varargin{2:end});
         else
            % For randi(max,m,n,...,'ClassName')
            if numel([varargin{:}]) < 2
               error('Not enough input arguments')
            end
            dims = [varargin{2:end}];
            r = zeros(dims,'Color');
```

```
            for k = 1:prod(dims)
               r(k) = Color('RGB',randi(varargin{1},[1,3]));
            end
         end
      end
   end
end
```

Prototype Object Method Implementation

The objective of a method that returns an array of objects that are “like a prototype object” depends on the requirements of the class. For the `Color` class, the `zeroLike` method creates objects that have the `ColorSpace` property value of the prototype object, but the `ColorValues` are all zero.

Here is an implementation of a `zerosLike` method for the `Color` class. This implementation:

- Defines the `zerosLike` method as `Hidden`
- Returns a scalar `Color` object if the call to the `zeros` function has no dimension arguments
- Returns an empty array if the call to the `zeros` function has any dimension arguments that are negative or equal to 0.
- Returns an array of `Color` objects of the dimensions specified by the call to the `zeros` function.

```
classdef Color
   ...
   methods (Hidden)
      function z = zerosLike(obj,varargin)
         if nargin == 1
            % For zeros('like',obj)
            cSpace = obj.ColorSpace;
            z = Color;
            z.ColorSpace = cSpace;
         elseif  any([varargin{:}] <= 0)
            % For zeros with any dimension <= 0
            z = Color.empty(varargin{:});
         else
            % For zeros(m,n,...,'like',obj)
            if ~isscalar(obj)
               error('Prototype object must be scalar')
            end
            obj = Color(obj.ColorSpace,zeros(1,3,'like',obj.ColorValues));
            z = repmat(obj,varargin{:});
         end
      end
   end
end
```

Full Class Listing

Here is the `Color` class definition with the overloaded methods.

Note In actual practice, the `Color` class requires error checking, color space conversions, and so on. This overly simplified version illustrates the implementation of the overloaded methods.

```
classdef Color
   properties
      ColorValues = [0,0,0]
      ColorSpace = 'RGB'
   end
   methods
      function obj = Color(cSpace,values)
```

```
            if nargin > 0
               obj.ColorSpace = cSpace;
               obj.ColorValues = values;
            end
         end
      end
      methods (Static)
         function z = zeros(varargin)
            if (nargin == 0)
               % For zeros('ClassName')
               z = Color;
            elseif any([varargin{:}] <= 0)
               % For zeros with any dimension <= 0
               z = Color.empty(varargin{:});
            else
               % For zeros(m,n,...,'ClassName')
               % Use property default values
               z = repmat(Color,varargin{:});
            end
         end
         function r = randi(varargin)
            if (nargin == 0)
               % For randi('ClassName')
               r = Color('RGB',randi(255,[1,3]));
            elseif any([varargin{2:end}] <= 0)
               % For randi with any dimension <= 0
               r = Color.empty(varargin{2:end});
            else
               % For randi(max,m,n,...,'ClassName')
               if numel([varargin{:}]) < 2
                  error('Not enough input arguments')
               end
               dims = [varargin{2:end}];
               r = zeros(dims,'Color');
               for k = 1:prod(dims)
                  r(k) = Color('RGB',randi(varargin{1},[1,3]));
               end
            end
         end
      end
      methods (Hidden)
         function z = zerosLike(obj,varargin)
            if nargin == 1
               % For zeros('like',obj)
               cSpace = obj.ColorSpace;
               z = Color;
               z.ColorSpace = cSpace;
            elseif  any([varargin{:}] <= 0)
               % For zeros with any dimension <= 0
               z = Color.empty(varargin{:});
            else
               % For zeros(m,n,...,'like',obj)
               if ~isscalar(obj)
                  error('Prototype object must be scalar')
               end
               obj = Color(obj.ColorSpace,zeros(1,3,'like',obj.ColorValues));
               z = repmat(obj,varargin{:});
            end
         end
      end
   end
```

See Also

Related Examples

- “Construct Object Arrays” on page 10-2

Class Methods for Graphics Callbacks

In this section...

Referencing the Method

To use an ordinary class method as callback for a graphics object, specify the callback property as a function handle referencing the method. For example,

```
uicontrol('Style','slider','Callback',@obj.sliderCallback)
```

Where your class defines a method called `sliderCallback` and `obj` is an instance of your class.

To use a static method as a callback, specify the callback property as a function handle that includes the class name that is required to refer to a static method:

```
uicontrol('Style','slider','Callback',@MyClass.sliderCallback)
```

Syntax for Method Callbacks

For ordinary methods, use dot notation to pass an instance of the class defining the callback as the first argument:

```
@obj.methodName
```

Define the callback method with the following input arguments:

- An instance of the defining class as the first argument
- The event source handle
- The event data

The function signature would be of this form:

```
function methodName(obj,src,eventData)
   ...
end
```

For static methods, the required class name ensures MATLAB dispatches to the method of the specified class:

```
@MyClass.methodName
```

Define the static callback method with two input arguments — the event source handle and the event data

The function signature would be of this form:

```
function methodName(src,eventData)
```

Passing Extra Arguments

If you want to pass arguments to your callback in addition to the source and event data arguments passed by MATLAB, you can use an anonymous function. The basic syntax for an anonymous function that you assign to the graphic object's `Callback` property includes the object as the first argument:

```
@(src,event)callbackMethod(object,src,eventData,arg1,...argn)
```

The function signature would be of this form:

```
function methodName(obj,src,eventData,varargin)
   ...
end
```

Use a Class Method for a Slider Callback

This example shows how to use a method of your class as a callback for an uicontrol slider.

The `SeaLevelSlider` class creates a slider that varies the color limits of an indexed image to give the illusion of varying the sea level.

Class Definition

Define `SeaLevelSlider` as a handle class with the following members:

- The class properties store figure and axes handles and the calculated color limits.
- The class constructor creates the graphics objects and assigns the slider callback.
- The callback function for the slider accepts the three required arguments — a class instance, the handle of the event source, and the event data. The event data argument is empty and not used.
- The uicontrol callback uses dot notation to reference the callback method: `...'Callback',@obj.sliderCallback`.

```
classdef SeaLevelSlider < handle
   properties
      Figure
      Axes
      CLimit
   end

   methods
      function obj = SeaLevelSlider(x,map)
         obj.Figure = figure('Colormap',map,...
            'Position',[100,100,560,580],...
            'Resize','off');
         obj.Axes = axes('DataAspectRatio',[1,1,1],...
            'XLimMode','manual','YLimMode','manual',...
            'Parent',obj.Figure);
         image(x,'CDataMapping','scaled',...
            'Parent',obj.Axes);
         obj.CLimit = get(obj.Axes,'CLim');
         uicontrol('Style','slider',...
            'Parent',obj.Figure,...
            'Max',obj.CLimit(2)-10,...
            'Min',obj.CLimit(1)-1,...
            'Value',obj.CLimit(1),...
```

```
            'Units','normalized',...
            'Position',[0.9286,0.1724,0.0357,0.6897],...
            'SliderStep',[0.003,0.005],...
            'Callback',@obj.sliderCallback);
      end

      function sliderCallback(obj,src,~)
         minVal = get(src,'Value');
         maxVal = obj.CLimit(2);
         obj.Axes.CLim = [minVal maxVal];
      end
   end
end
```

Using the SeaLevelAdjuster Class

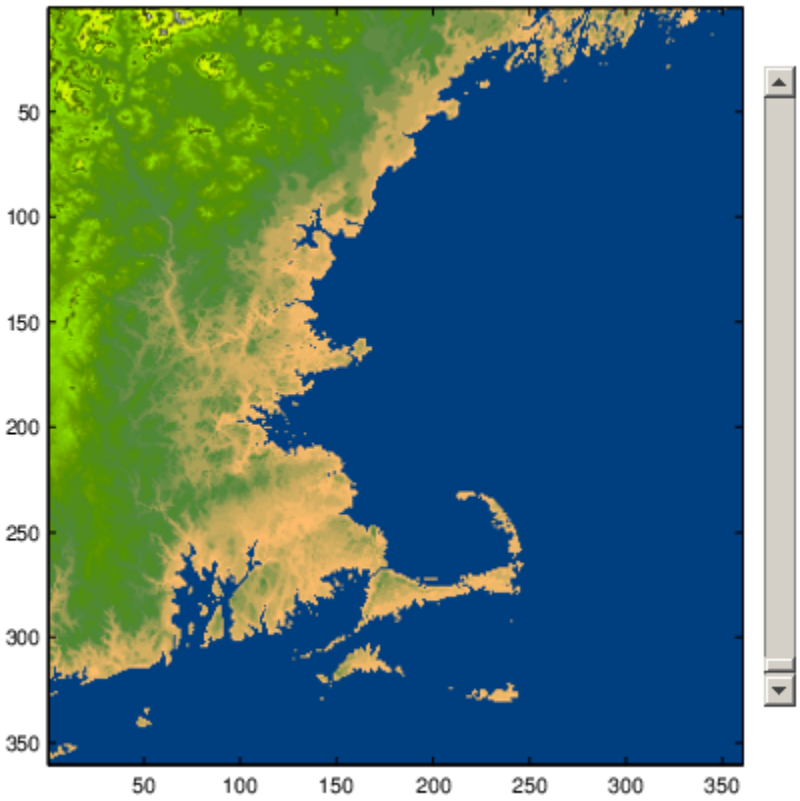
The class uses the `cape` image that is included with the MATLAB product. To obtain the image data, use the `load` command:

```
load cape X map
```

After loading the data, create a `SeaLevelSlider` object for the image:

```
slaObj = SeaLevelSlider(X,map);
```

Move the slider to change the color mapping and visualize a rise in sea level.



See Also

More About

- “Listener Callback Syntax” on page 11-24

Object Arrays

Construct Object Arrays

In this section...

Build Arrays in the Constructor

A class constructor can create an array by building the array and returning it as the output argument.

For example, the `ObjectArray` class creates an object array that is the same size as the input array. Then it initializes the `Value` property of each object to the corresponding input array value.

```
classdef ObjectArray
   properties
      Value
   end
   methods
      function obj = ObjectArray(F)
         if nargin ~= 0
            m = size(F,1);
            n = size(F,2);
            obj(m,n) = obj;
            for i = 1:m
               for j = 1:n
                  obj(i,j).Value = F(i,j);
               end
            end
         end
      end
   end
end
```

To preallocate the object array, assign the last element of the array first. MATLAB fills the first to penultimate array elements with the `ObjectArray` object.

After preallocating the array, assign each object `Value` property to the corresponding value in the input array `F`. To use the class:

- Create 5-by-5 array of magic square numbers
- Create a 5-by-5 object array

```
F = magic(5);
A = ObjectArray(F);
whos
```

```
  Name      Size            Bytes  Class          Attributes

  A         5x5               304  ObjectArray
  F         5x5               200  double
```

Referencing Property Values in Object Arrays

Given an object array `objArray` in which each object has a property `PropName`:

- Reference the property values of specific objects using array indexing:

  ```
  objArray(ix).PropName
  ```

- Reference all values of the same property in an object array using dot notation. MATLAB returns a comma-separated list of property values.

  ```
  objArray.PropName
  ```

- To assign the comma-separated list to a variable, enclose the right-side expression in brackets:

  ```
  values = [objArray.PropName]
  ```

For example, given the `ObjProp` class:

```
classdef ObjProp
   properties
      RegProp
   end
   methods
      function obj = ObjProp
         obj.RegProp = randi(100);
      end
   end
end
```

Create an array of `ObjProp` objects:

```
for k = 1:5
   objArray(k) = ObjProp;
end
```

Access the `RegProp` property of the second element of the object array using array indexing:

```
objArray(2).RegProp
```

```
ans =

    91
```

Assign the values of all `RegProp` properties to a numeric array:

```
propValues = [objArray.RegProp]
```

```
propValues =

    82    91    13    92    64
```

Use standard indexing operations to access the values of the numeric array. For more information on numeric arrays, see “Matrices and Arrays”.

See Also

Related Examples

- “Initialize Object Arrays” on page 10-5
- “Initialize Arrays of Handle Objects” on page 10-9

- “Class Constructor Methods” on page 9-15

Initialize Object Arrays

In this section...

Calls to Constructor

During the creation of object arrays, MATLAB can call the class constructor with no arguments, even if the constructor does not build an object array. For example, suppose that you define the following class:

```
classdef SimpleValue
   properties
      Value
   end
   methods
      function obj = SimpleValue(v)
         obj.Value = v;
      end
   end
end
```

Execute the following statement to create an array:

```
a(1,7) = SimpleValue(7)
```

```
Error using SimpleValue (line 7)
Not enough input arguments.
```

This error occurs because MATLAB calls the constructor with no arguments to initialize elements `1` through `6` in the array.

Your class must support the no input argument constructor syntax. A simple solution is to test `nargin` and let the case when `nargin == 0` execute no code, but not error:

```
classdef SimpleValue
   properties
      Value
   end
   methods
      function obj = SimpleValue(v)
         if nargin > 0
            obj.Value = v;
         end
      end
   end
end
```

Using the revised class definition, the previous array assignment statement executes without error:

```
a(1,7) = SimpleValue(7)
```

```
a =
```

```
  1x7 SimpleValue array with properties:

    Value
```

The object assigned to array element `a(1,7)` uses the input argument passed to the constructor as the value assigned to the property:

```
a(1,7)

ans =
  SimpleValue with properties:

    Value: 7
```

MATLAB created the objects contained in elements `a(1,1:6)` with no input argument. The default value for properties empty `[]`. For example:

```
a(1,1)

ans =
  SimpleValue with properties:

    Value: []
```

MATLAB calls the `SimpleValue` constructor once and copies the returned object to each element of the array.

Initial Value of Object Properties

When MATLAB calls a constructor with no arguments to initialize an object array, one of the following assignments occurs:

- If property definitions specify default values, MATLAB assigns these values.
- If the constructor assigns values in the absence of input arguments, MATLAB assigns these values.
- If neither of the preceding situations apply, MATLAB assigns the value of empty double (that is, `[]`) to the property.

See Also

Related Examples

- “Initialize Arrays of Handle Objects” on page 10-9

Empty Arrays

In this section...

Creating Empty Arrays

Empty arrays have no elements, but are of a certain class. All nonabstract classes have a static method named `empty` that creates an empty array of the same class. The `empty` method enables you to specify the dimensions of the output array. However, at least one of the dimensions must be `0`. For example, define the `SimpleValue` class:

```
classdef SimpleValue
   properties
      Value
   end
   methods
      function obj = SimpleValue(v)
         if nargin > 0
            obj.Value = v;
         end
      end
   end
end
```

Create a 5–by–0 empty array of class `SimpleValue`.

```
ary = SimpleValue.empty(5,0)
```

```
ary =

  5x0 SimpleValue array with properties:

    Value
```

Calling `empty` with no arguments returns a 0–by–0 empty array.

Assigning Values to an Empty Array

An empty object defines the class of an array. To assign nonempty objects to an empty array, MATLAB calls the class constructor to create default instances of the class for every other array element. Once you assign a nonempty object to an array, all array elements must be nonempty objects.

Note A class constructor must avoid returning empty objects by default.

For example, using the `SimpleValue` defined in the “Initialize Object Arrays” on page 10-5 section, create an empty array:

```
ary = SimpleValue.empty(5,0);
class(ary)
```

```
ans =

SimpleValue
```

`ary` is an array of class `SimpleValue`. However, it is an empty array:

```
ary(1)
```

```
Index exceeds matrix dimensions.
```

If you make an assignment to a property value, MATLAB calls the `SimpleClass` constructor to grow the array to the require size:

```
ary(5).Value = 7;
ary(5).Value
```

```
ans =

    7
```

```
ary(1).Value
```

```
ans =

    []
```

MATLAB populates array elements one through five with `SimpleValue` objects created by calling the class constructor with no arguments. Then MATLAB assigns the property value 7 to the object at `ary(5)`.

See Also

Related Examples

- “Initialize Arrays of Handle Objects” on page 10-9

Initialize Arrays of Handle Objects

When initializing an array of handle objects, MATLAB fills in the empty elements of an array with a default object. To create the default object, MATLAB:

- Calls the class constructor once to obtain an object
- Creates unique handles for each element in the array
- Copies the property values from the constructed default object without calling the constructor again.

The `InitHandleArray` class illustrates this behavior.

```
classdef InitHandleArray < handle
   properties
      RandNumb
   end
   methods
      function obj = InitHandleArray
         obj.RandNumb = randi(100);
      end
   end
end
```

The property `RandNumb` contains a random number that the `InitHandleArray` constructor assigns.

Consider what happens when MATLAB initializes an array created by assigning to the last element in the array. (The last element is the one with the highest index values). Suppose the value of the `RandNumb` property of the `InitHandleArray` object assigned to the element `A(4,5)` is 59:

```
A(4,5) = InitHandleArray;
A(4,5).RandNumb
```

```
ans =

     59
```

The element in the index location `A(4,5)` is an instance of the `InitHandleArray` class. The default object used for element `A(1,1)` is also an instance of the `InitHandleArray` class, but its `RandNumb` property is set to a different random number.

To fill in the preceding array elements, MATLAB calls the class constructor to create a single object. MATLAB copies this object to all the remaining array elements. Calling the constructor to create the default object resulted in another call to the `randi` function, which returns a new random number:

```
A(1,1).RandNumb
```

```
ans =

     10
```

MATLAB copies this second instance to all remaining array elements:

```
A(2,2).RandNumb
```

```
ans =

     10
```

```
A(2,3).RandNumb

ans =

    10
```

When initializing an object array, MATLAB assigns a copy of a single object to the empty elements in the array. MATLAB gives each object a unique handle so that later you can assign different property values to each object. The objects are not equivalent:

```
A(1,1) == A(2,2)

ans =

     0
```

That is, the handle `A(1,1)` does not refer to the same object as `A(2,2)`. The creation of an array with a statement such as:

```
A(4,5) = InitHandleArray;
```

results in two calls to the class constructor. The first creates the object for array element `A(4,5)`. The second creates a default object that MATLAB copies to all remaining empty array elements.

Related Information

For information on array manipulation, see “Multidimensional Arrays”

See “Handle Objects as Default Property Values” on page 8-14 for information on assigning values to properties.

See “Customize Object Indexing” on page 17-7 for information on implementing `subsasgn` methods for your class.

Accessing Dynamic Properties in Arrays

You cannot reference all the dynamic properties in an object array using a single statement, as you can with ordinary properties. For example, the `ObjectArrayDynamic` class subclasses the `dynamicprops` class.

```
classdef ObjectArrayDynamic < dynamicprops
   properties
      RegProp
   end
   methods
      function obj = ObjectArrayDynamic
         obj.RegProp = randi(100);
      end
   end
end
```

You can add dynamic properties to objects of the `ObjectArrayDynamic` class. Create an object array and add dynamic properties to each member of the array. Define elements 1 and 2 as `ObjectArrayDynamic` objects:

```
a(1) = ObjectArrayDynamic;
a(2) = ObjectArrayDynamic;
```

Add dynamic properties to each object and assign a value.

```
a(1).addprop('DynoProp');
a(1).DynoProp = 1;
a(2).addprop('DynoProp');
a(2).DynoProp = 2;
```

Get the values of the ordinary properties, as with any array.

```
a.RegProp
```

```
ans =

     4

ans =

    85
```

However, MATLAB returns an error if you try to access the dynamic properties of all array elements using this syntax.

```
a.DynoProp
```

```
No appropriate method, property, or field 'DynoProp' for class
'ObjectArrayDynamic'.
```

Refer to each object individually to access dynamic property values:

```
a(1).DynoProp
```

```
ans =

     1
```

```
a(2).DynoProp

ans =

     2
```

For information about classes that can define dynamic properties, see “Dynamic Properties — Adding Properties to an Instance” on page 8-47 .

Implicit Class Conversion

In this section...

MATLAB can implicitly convert classes in these situations:

- Creation or modification of object arrays using concatenation
- Creation or modification of object arrays using subscripted assignment
- Property validation
- Argument validation in function and method calls

To perform the conversion, MATLAB attempts to use a converter method, the constructor of the destination class, or the `cast` function, depending on the context of the conversion.

Concatenation

In concatenation operations, the dominant object determines the class of the resulting array. MATLAB determines the dominant object according to these rules:

- User-defined classes are dominant over built-in classes such as `double`.
- If there is no defined dominance relationship between any two classes, then the leftmost object in the concatenation statement is dominant. For more information on class dominance, see “Method Invocation” on page 9-11.

For example, `A` is an instance of `ClassA`, and `B` is an instance of `ClassB`. In the statement `C = [A,B]`, if `A` is the dominant object, MATLAB attempts to convert `B` to the class of `A`.

MATLAB first tries to call a converter method. If no converter method is found, it calls the constructor for `ClassA`. The concatenation statement is equivalent to:

```
C = [A,ClassA(B)]
```

If the call to the `ClassA` constructor fails to convert `B` to `ClassA`, MATLAB issues an error. If the conversion succeeds, MATLAB concatenates `A` with the converted `B`.

Subscripted Assignment

In subscripted assignment, the left side of the assignment statement defines the class of the array. If you assign array elements when the right side is a different class than the left side, MATLAB attempts to convert the right side to the class of the left side.

For example, assigning an object of `ClassB` to an element of array `A` requires conversion.

```
A = ClassA;
B = ClassB;
A(2) = B;
```

MATLAB attempts to perform the conversion by first looking for a converter method. If no converter method is found, it calls the `ClassA` constructor. The assignment is then equivalent to:

```
A(2) = ClassA(B)
```

If calling the constructor fails, MATLAB calls `cast`:

```
A(2) = cast(B,"like",A)
```

If the conversion still fails after these steps, MATLAB issues an error. If the conversion succeeds, MATLAB assigns the converted value to `A(2)`.

Property Validation

When you assign a value to a property that specifies a class restriction as part of its validation, MATLAB uses the built-in function `isa` to check the relationship between the value and the class. If the value is not the specified class or one of its subclasses, MATLAB attempts to convert the value to the specified class.

To demonstrate the conversion process, refer to these class definitions.

```
classdef ClassA
   properties
     Prop ClassB
   end
end

classdef ClassB
end

classdef SubClassB < ClassB
end

classdef ClassC
end
```

This script shows when MATLAB attempts conversions during property assignments.

```
A = ClassA;
B = ClassB;
SB = SubClassB;
C = ClassC;

A.Prop = B; % no conversion
A.Prop = SB; % no conversion
A.Prop = C; % conversion required
```

In this script:

- `A.Prop = B` does not require conversion because `B` belongs to `ClassB`, which satisfies the property validation for `Prop` defined in `ClassA`.
- `A.Prop = SB` does not require conversion because `SB` belongs to `SubClassB`, which is a subclass of `ClassB`.
- `A.Prop = C` requires conversion because `C` does not belong to `ClassB` or its subclass `SubClassB`.

MATLAB attempts to convert `C` to `ClassB` or its subclass `SubClassB` by first calling `ClassB(C)`. This call can invoke a converter method called `ClassB` (defined by `ClassC`) or the `ClassB` constructor. If calling `ClassB(C)` does not yield an instance of `ClassB` or `SubClassB`, MATLAB attempts to cast the result of `ClassB(C)` to `ClassB` as a final step.

If these steps fail to convert `C` to `ClassB` or `SubClassB`, MATLAB issues an error. If the conversion succeeds, MATLAB writes the converted value to `A.Prop`.

Note Property class validation does not support implicit conversion of any built-in types to cell arrays, even if you provide your own conversion method.

Function and Method Argument Validation

When you assign a value to a function or method argument that specifies a class restriction as part of its validation, MATLAB uses the built-in function `isa` to check the relationship between the value and the class. If the value is not the specified class or one of its subclasses, MATLAB attempts to convert the value to the specified class.

To demonstrate the conversion process, refer to these class and function definitions.

```
classdef ClassA
end

classdef SubClassA < ClassA
end

classdef ClassB
end

function test(x)
   arguments
      x ClassA
   end
end
```

This script shows when MATLAB attempts conversions during argument validation.

```
A = ClassA;
SA = SubClassA;
B = ClassB;

test(A); % no conversion
test(SA); % no conversion
test(B); % conversion required
```

In this script:

- `test(A)` does not require conversion because `A` belongs to `ClassA`, which satisfies the argument validation for `x` defined in `test`.
- `test(SA)` does not require conversion because `SA` belongs to `SubClassA`, which is a subclass of `ClassA`.
- `test(B)` requires conversion because `B` does not belong to `ClassA` or its subclass `SubClassA`.

  MATLAB attempts to convert `B` to `ClassA` or its subclass `SubClassA` by first calling `ClassA(B)`. This call can invoke a converter method called `ClassA` (defined by `ClassB`) or the `ClassA`

constructor. If calling `ClassA(B)` does not yield an instance of `ClassA` or `SubClassA`, MATLAB attempts to cast the result of `ClassA(B)` to `ClassA` as a final step.

If these steps fail to convert `B` to `ClassA` or `SubClassA`, MATLAB issues an error. If the conversion succeeds, MATLAB uses the converted value for the function call.

Note Argument validation does not support implicit conversion of any built-in types to cell arrays, even if you provide your own conversion method.

See Also

Related Examples

- “Valid Combinations of Unlike Classes”
- “Concatenating Objects of Different Classes” on page 10-17
- “Object Converters” on page 17-5
- “Function Argument Validation”

Concatenating Objects of Different Classes

In this section...

Basic Knowledge

The material presented in this section builds on an understanding of the information presented in the following sections.

- “Construct Object Arrays” on page 10-2
- “Valid Combinations of Unlike Classes”

MATLAB Concatenation Rules

MATLAB follows these rules for concatenating objects:

- MATLAB always attempts to convert all objects to the dominant class.
- User-defined classes take precedence over built-in classes like `double`.
- If there is no defined dominance relationship between any two objects, then the leftmost object dominates (see “Class Precedence” on page 6-19).

When converting to a dominant class during concatenation or subscripted assignment, MATLAB searches the non-dominant class for a conversion method that is the same name as the dominant class. If such a conversion method exists, MATLAB calls it. If a conversion method does not exist, MATLAB calls the dominant class constructor on the non-dominant object.

It is possible for the dominant class to define `horzcat`, `vertcat`, or `cat` methods that modify the default concatenation process.

Note MATLAB does not convert objects to a common superclass unless those objects are part of a heterogeneous hierarchy. For more information, see “Designing Heterogeneous Class Hierarchies” on page 10-22.

Concatenating Objects

Concatenation combines objects into arrays. For example:

```
ary = [obj1,obj2,obj3];
```

The size of `ary` is 1-by-3.

The class of the arrays is the same as the class of the objects being concatenated. Concatenating objects of different classes is possible if MATLAB can convert objects to the dominant class. MATLAB attempts to convert unlike objects by:

- Calling the inferior object converter method, if one exists.
- Passing an inferior object to the dominant class constructor to create an object of the dominant class.

If conversion of the inferior object is successful, MATLAB returns an array that is of the dominant class. If conversion is not possible, MATLAB returns an error.

Calling the Dominant-Class Constructor

MATLAB calls the dominant class constructor to convert an object of an inferior class to the dominant class. MATLAB passes the inferior object to the constructor as an argument. If the class design enables the dominant class constructor to accept objects of inferior classes as input arguments, then concatenation is possible without implementing a separate converter method.

If the constructor simply assigns this argument to a property, the result is an object of the dominant class with an object of an inferior class stored in a property. If this assignment is not a desired result, then ensure that class constructors include adequate error checking.

For example, consider the class `ColorClass` and two subclasses, `RGBColor` and `HSVColor`:

```
classdef ColorClass
   properties
      Color
   end
end
```

The class `RGBColor` inherits the `Color` property from `ColorClass`. `RGBColor` stores a color value defined as a three-element vector of red, green, and blue (RGB) values. The constructor does not restrict the value of the input argument. It assigns this value directly to the `Color` property.

```
classdef RGBColor < ColorClass
   methods
      function obj = RGBColor(rgb)
         if nargin > 0
            obj.Color = rgb;
         end
      end
   end
end
```

The class `HSVColor` also inherits the `Color` property from `ColorClass`. `HSVColor` stores a color value defined as a three-element vector of hue, saturation, brightness value (HSV) values.

```
classdef HSVColor < ColorClass
   methods
      function obj = HSVColor(hsv)
         if nargin > 0
            obj.Color = hsv;
         end
      end
   end
end
```

Create an instance of each class and concatenate them into an array. The `RGBColor` object is dominant because it is the leftmost object and neither class defines a dominance relationship:

```
crgb = RGBColor([1 0 0]);
chsv = HSVColor([0 1 1]);
ary = [crgb,chsv];
class(ary)
```

```
ans =

RGBColor
```

You can combine these objects into an array because MATLAB can pass the inferior object of class `HSVColor` to the constructor of the dominant class. However, notice that the `Color` property of the second `RGBColor` object in the array actually contains an `HSVColor` object, not an `RGB` color specification:

```
ary(2).Color
```

```
ans =

  HSVColor with properties:

    Color: [0 1 1]
```

Avoid this undesirable behavior by:

- Implementing converter methods
- Performing argument checking in class constructors before assigning values to properties

Converter Methods

If your class design requires object conversion, implement converter methods for this purpose.

The `ColorClass` class defines converter methods for `RGBColor` and `HSVColor` objects:

```
classdef ColorClass
   properties
      Color
   end
   methods
      function rgbObj = RGBColor(obj)
         if isa(obj,'HSVColor')
            rgbObj = RGBColor(hsv2rgb(obj.Color));
         end
      end
      function hsvObj = HSVColor(obj)
         if isa(obj,'RGBColor')
            hsvObj = HSVColor(rgb2hsv(obj.Color));
         end
      end
   end
end
```

Create an array of `RGBColor` and `HSVColor` objects with the revised superclass:

```
crgb = RGBColor([1 0 0]);
chsv = HSVColor([0 1 1]);
```

```
ary = [crgb,chsv];
class(ary)

ans =

RGBColor
```

MATLAB calls the converter method for the `HSVColor` object, which it inherits from the superclass. The second array element is now an `RGBColor` object with an RGB color specification assigned to the `Color` property:

```
ary(2)

ans =

  RGBColor with properties:

    Color: [1 0 0]

ary(2).Color

ans =

     1     0     0
```

If the leftmost object is of class `HSVColor`, the array `ary` is also of class `HSVColor`, and MATLAB converts the `Color` property data to HSV color specification.

```
ary = [chsv crgb]

ary =

  1x2 HSVColor

  Properties:
    Color

ary(2).Color

ans =

     0     1     1
```

Defining a converter method in the superclass and adding better argument checking in the subclass constructors produces more predictable results. Here is the `RGBColor` class constructor with argument checking:

```
classdef RGBColor < ColorClass
   methods
      function obj = RGBColor(rgb)
         if nargin == 0
            rgb = [0 0 0];
         else
            if ~(isa(rgb,'double')...
                  && size(rgb,2) == 3 ...
                  && max(rgb) <= 1 && min(rgb) >= 0)
               error('Specify color as RGB values')
            end
         end
```

```
            obj.Color = rgb;
        end
    end
end
```

Your applications can require additional error checking and other coding techniques. The classes in these examples are designed only to demonstrate concepts.

See Also

More About

- “Implicit Class Conversion” on page 10-13
- “Object Converters” on page 17-5
- “Hierarchies of Classes — Concepts” on page 12-2

Designing Heterogeneous Class Hierarchies

In this section...

Creating Classes That Support Heterogeneous Arrays

This topic describes the concepts involved in defining classes that support the formation of heterogeneous arrays. For information on the concatenation of existing MATLAB objects, see these topics.

- “Concatenating Objects of Different Classes” on page 10-17
- “Valid Combinations of Unlike Classes”

For an example that uses heterogeneous arrays, see “A Class Hierarchy for Heterogeneous Arrays” on page 20-2.

MATLAB Arrays

MATLAB determines the class of an array by the class of the objects contained in the array. MATLAB is unlike some languages in which you define an array of object pointers or references. In these other languages, the type of the array is different from the type of an object in the array. You can access the elements of the array and dispatch to methods on those elements, but you cannot call an object method on the whole array, as you can in MATLAB.

Object arrays in MATLAB are homogeneous in class. Because of this homogeneity, you can perform operations on whole arrays, such as multiplying numeric matrices. You can form heterogeneous arrays by defining a hierarchy of classes that derive from a common superclass. Cell arrays provide option for an array type that can hold different kinds of unrelated objects.

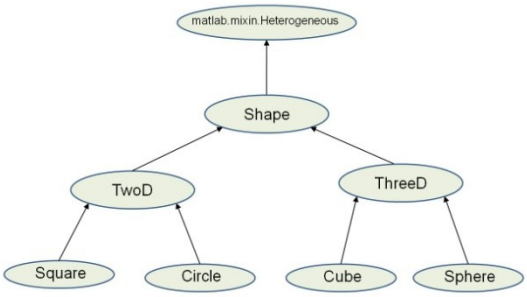
Heterogeneous Hierarchies

You can form arrays of objects that are subclasses of a common superclass when these classes are part of a heterogeneous hierarchy. A MATLAB heterogeneous class hierarchy:

- Derives from `matlab.mixin.Heterogeneous`

- Defines a single root superclass that derives directly from `matlab.mixin.Heterogeneous`
- Seals methods that are inherited by subclasses.

For example, in the following diagram, `Shape` is the root of the heterogeneous hierarchy.



Heterogeneous Arrays

A heterogeneous array is an array of objects that differ in their specific class, but all objects derive from or are instances of a common superclass. The common superclass forms the root of the hierarchy of classes that you can combine into heterogeneous arrays.

The common superclass must derive from `matlab.mixin.Heterogeneous`. Methods that you can call on the array as a whole must have the same definitions for all subclasses.

Heterogeneous hierarchies are useful to:

- Create arrays of objects that are of different classes, but part of a related hierarchy.
- Call methods of the most specific common superclass on the array as a whole
- Access properties of the most specific common superclass using dot notation with the array
- Use common operators that are supported for object arrays
- Support array indexing (scalar or nonscalar) that returns arrays of the most specific class

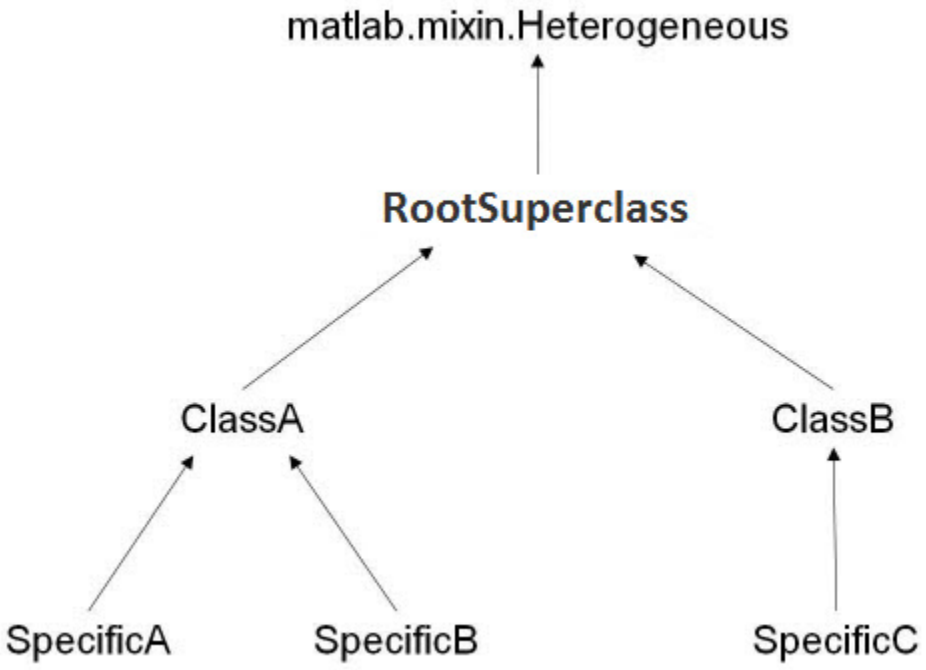
Heterogeneous Array Concepts

- Heterogeneous array — An array in which two or more elements belong to different specific classes. All elements derive from the same root superclass.
- Root superclass — Class derived directly from `matlab.mixin.Heterogeneous`. The root superclass can be abstract or concrete. Only concrete subclasses of the root superclass can form heterogeneous arrays.
- Most specific common superclass — The most specific class in the inheritance hierarchy from which all the objects in a heterogeneous array derive. The most specific common superclass can be the root superclass or a more specific superclass shared by the objects currently in the array.
- Class of a heterogeneous array — The most specific common superclass from which all objects in the heterogeneous array derive. Adding and removing objects from a heterogeneous array can change the most specific superclass shared by the instances. This change results in a change in the class of a heterogeneous array. The most specific common superclass can be abstract.

Nature of Heterogeneous Arrays

The heterogeneous hierarchy in this diagram illustrates the characteristics of heterogeneous arrays concerning:

- Array class
- Property access
- Method invocation



Class of Heterogeneous Arrays

The class of a heterogeneous array is that of the most specific superclass shared by the objects of the array.

If the following conditions are true, the concatenation and subscripted assignment operations return a heterogeneous array:

- The objects on the right side of the assignment statement are of different classes
- All objects on the right side of the assignment statement derive from a common subclass of `matlab.mixin.Heterogeneous`

For example, form an array by concatenating objects of these classes. The class of `a1` is `ClassA`:

```
a1 = [SpecificA,SpecificB];
class(a1)
```

```
ans =

ClassA
```

If the array includes an object of the class `SpecificC`, the class of `a2` is `RootSuperclass`:

```
a2 = [SpecificA,SpecificB,SpecificC];
class(a2)

ans =

RootSuperclass
```

If you assigned an object of the class `SpecificC` to array `a1` using indexing, the class of `a1` becomes `RootSuperclass`:

```
a1(3) = SpecificC;
class(a1)

ans =

RootSuperclass
```

If the array contains objects of only one class, then the array is not heterogeneous. For example, the class of `a` is `SpecificA`.

```
a = [SpecificA,SpecificA];
class(a)

ans =

SpecificA
```

Property Access

Access array properties with dot notation when the class of the array defines the properties. The class of the array is the most specific common superclass, which ensures all objects inherit the same properties.

For example, suppose `ClassA` defines a property called `Prop1`.

```
a1 = [SpecificA,SpecificB];
a1.Prop1
```

Referring to `Prop1` using dot notation returns the value of `Prop1` for each object in the array.

Invoking Methods

To invoke a method on a heterogeneous array, the class of the array must define or inherit the method as `Sealed`. For example, suppose `RootSuperclass` defines a `Sealed` method called `superMethod`.

Call the method on all objects in the array `a2`:

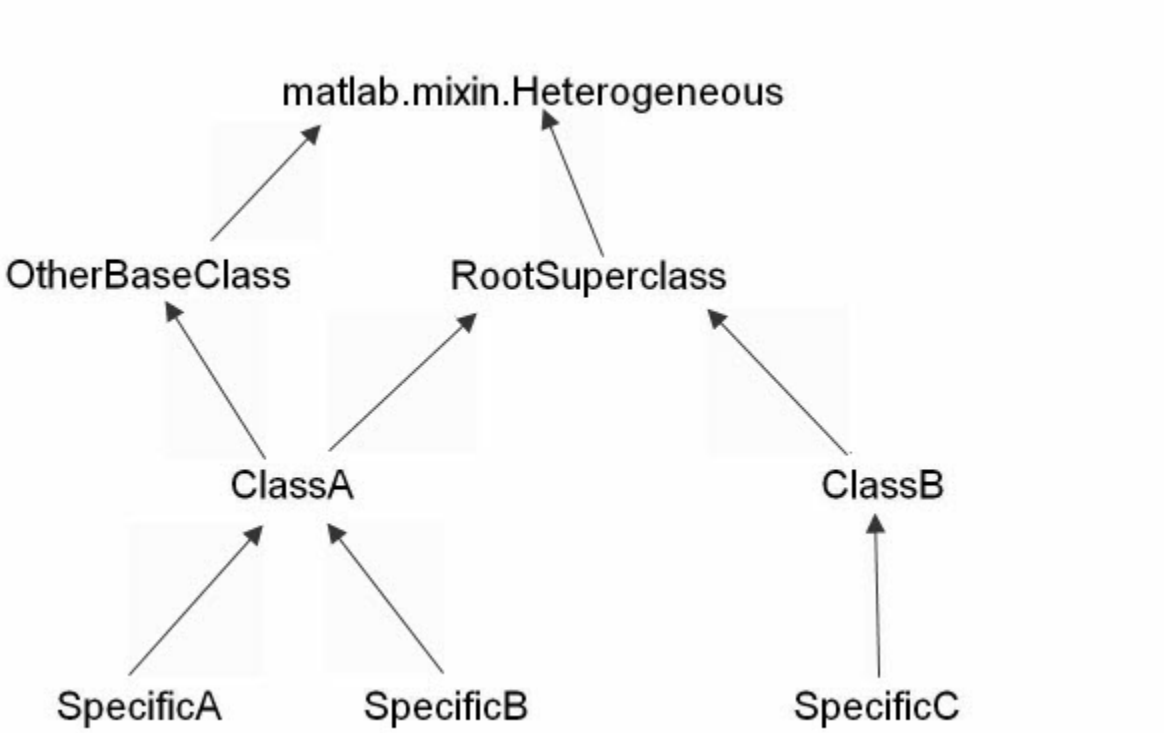
```
a2 = [SpecificA,SpecificB,SpecificC];
a2.superMethod
```

Sealing the method (so that it cannot be overridden in a subclass) ensures that the same method definition exists for all elements of the array. Calling that method on a single element of the array invokes the same method implementation as calling the method on the whole array.

Unsupported Hierarchies

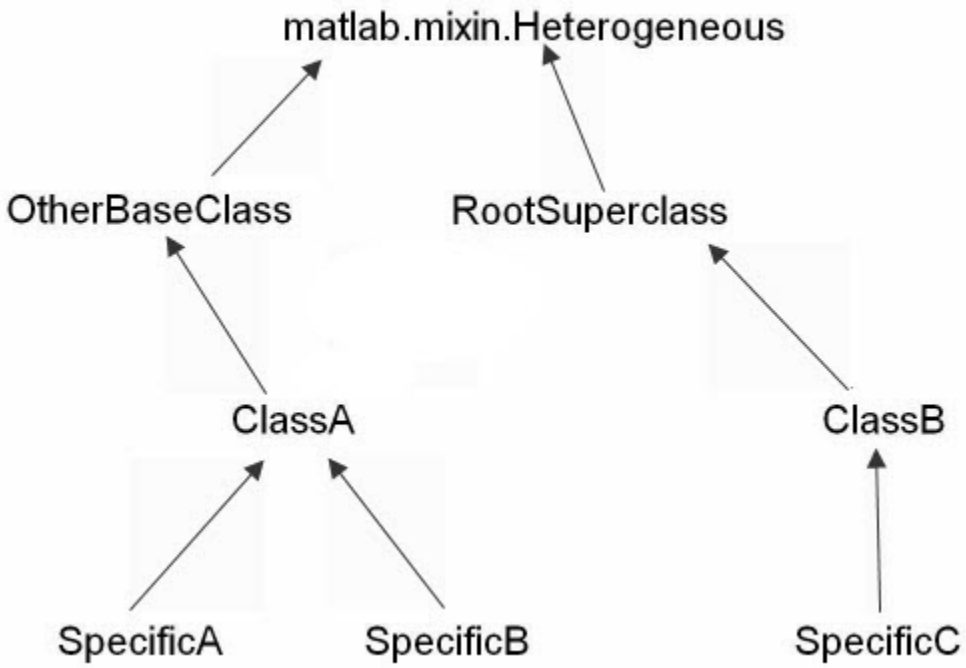
Heterogeneous hierarchies cannot have ambiguities when obtaining default objects, determining the class of the array, and converting class objects to other types. Members of the hierarchy can derive from only one root superclass (that is, from only one direct subclass of `matlab.mixin.Heterogeneous`).

This diagram shows a hierarchy that is not allowed:



`ClassA` derives from two classes that are subclasses of `matlab.mixin.Heterogeneous`.

The next diagram shows two separate heterogeneous hierarchies. `ClassA` has only one root superclass (called `OtherBaseClass`). The heterogeneous hierarchy is no longer ambiguous:



Default Object

A default object is the object returned by calling the class constructor with no arguments. MATLAB uses default objects in these situations:

- Indexed assignment creates an array with gaps in array elements. For example, assign the first element of array `h` to index `5`:

  ```
  h(5) = ClassA(arg1,arg2);
  ```

  MATLAB fills the unassigned positions with default objects.
- Loading a heterogeneous array from a MAT-file when the class definition of a specific object in the array is not available. MATLAB replaces the object with the default object.

Heterogeneous hierarchies enable you to define the default object for that hierarchy. The `matlab.mixin.Heterogeneous` class provides a default implementation of a method called `getDefaultScalarElement`. This method returns an instance of the root class of the heterogeneous hierarchy, unless the root superclass is abstract.

If the root superclass is abstract or is not appropriate for a default object, override the `getDefaultScalarElement` method. Implement the `getDefaultScalarElement` override in the root superclass, which derives directly from `matlab.mixin.Heterogeneous`.

`getDefaultScalarElement` must return a scalar object that is derived from the root superclass. For specific information on how to implement this method, see `matlab.mixin.Heterogeneous.getDefaultScalarElement`.

Conversion During Assignment and Concatenation

If you create a heterogeneous array that contains objects that are not derived from the same root superclass, MATLAB attempts to call a method called `convertObject`. Implement `convertObject` to convert objects to the appropriate class. There is no default implementation of this method.

To support the formation of heterogeneous arrays using objects that are not part of the heterogeneous hierarchy, implement a `convertObject` method in the root superclass. The `convertObject` method must convert the nonmember object to a valid member of the heterogeneous hierarchy.

For details on implementing the `convertObject` method, see `matlab.mixin.Heterogeneous`.

Empty Arrays of Heterogeneous Abstract Classes

For homogeneous arrays, MATLAB does not allow you to initialize an empty array of an abstract class. However, if the class is part of a heterogeneous hierarchy, you can initialize empty arrays of an abstract class. Initializing an empty heterogeneous array is useful in cases in which you do not know the class of the concrete elements in advance.

For example, suppose `RootSuperclass` is an abstract class that is the root of a heterogeneous hierarchy. Initialize an array using the `empty` static method:

```
ary = RootSuperclass.empty;
```

See Also

Related Examples

- “A Class Hierarchy for Heterogeneous Arrays” on page 20-2
- “Handle-Compatible Classes and Heterogeneous Arrays” on page 12-38

Heterogeneous Array Constructors

In this section...

Building Arrays in Superclass Constructors

When a subclass in a heterogeneous class hierarchy calls its superclass to construct an array of objects, you must ensure that the superclass constructor does not return a heterogeneous array to the subclass. The following programming patterns show how to avoid the errors caused by returning the wrong class to the subclass constructor.

When Errors Can Occur

Constructors must return objects that are the same class as the defining class. When working with objects from a heterogeneous class hierarchy, the class of an object array can change as you add array elements of different classes. As a result, heterogeneous superclass constructors can change the class of object arrays when the class design requires all of these techniques:

- Building object arrays in subclass constructors
- Calling superclass constructors from subclass constructors to pass arguments
- Creating object arrays in the superclass constructor

In addition, either of the following is true:

- The root superclass is not abstract and does not implement a `matlab.mixin.Heterogeneous.getDefaultScalarElement` method.
- The root superclass implements a `getDefaultScalarElement` method that returns an object that is not the same class as the subclass.

When assigning to object arrays, MATLAB uses the default object to fill in unassigned array elements. In a heterogeneous hierarchy, the default object can be the superclass that is called by the subclass constructor. Therefore, building an array in the superclass constructor can create a heterogeneous array.

If a superclass constructor returns a heterogeneous array to the subclass constructor, MATLAB generates an error (see “Potential Error” on page 10-32).

Initialize Array in Superclass Constructor

To avoid errors, initialize the object array explicitly in the superclass constructor. For example, use `repelem` in the superclass constructor to initialize the array before initializing the superclass part of the objects. Initializing the array ensures that all elements assigned into the array are of the same class as the `obj` argument.

In this code, the superclass constructor creates one object for each element in the input argument, `arg`:

```
method
   function obj = SuperClass(arg)
      ...
   n = numel(arg);
   obj = repelem(obj,1,n);
   for k = 1:n
      obj(k).SuperProp = arg(k);
   end
      ...
   end
end
```

The subclass constructor calls the superclass constructor to pass the required argument array, `a`:

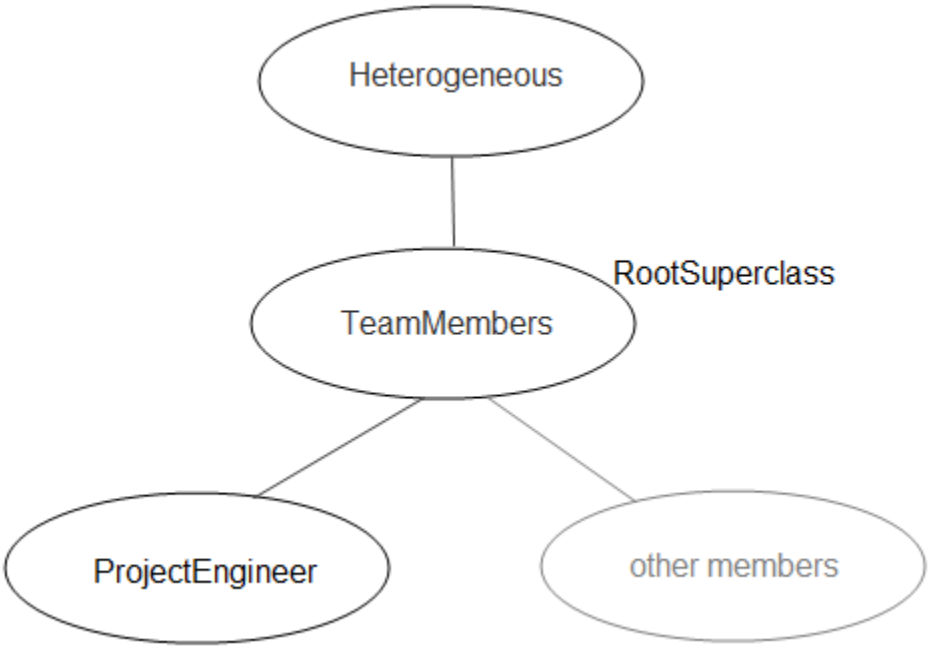
```
method
   function obj = SubClass(a)
         obj = obj@SuperClass(a);
         for k = 1:numel(a)
            obj(k).SubProp = a(k);
      end
   end
end
```

Sample Implementation

The following class hierarchy defines a subclass that builds object arrays in its constructor. The root superclass of the hierarchy initializes the superclass part of the objects in the array.

This class hierarchy represents members of an engineering team. The classes in the hierarchy include:

- `TeamMembers` — Superclass for specific team member classes, like `ProjectEngineer`. `TeamMembers` defines the `Name` and `PhoneX` properties and derives from `matlab.mixin.Heterogeneous`.
- `ProjectEngineer` — Team members that are engineers. Each instance inherits a `Name` and `PhoneX` property and defines a billing `Rate` property.
- Other members — Other types of team members not implemented for this example for simplicity.



The `TeamMembers` class is the root of the heterogeneous hierarchy and is a concrete class. Before assigning values to the `Name` and `PhoneX` properties, the constructor initializes an array of subclass (`ProjectEngineer`) objects.

The `ProjectEngineer` constructor provides the `obj` argument for the call to `repelem` with this statement:

```
obj = obj@TeamMembers(varargin{1:2});
```

Here is the `TeamMembers` class:

```
classdef TeamMembers < matlab.mixin.Heterogeneous
   properties
      Name
      PhoneX
   end
   methods
      function obj = TeamMembers(nme,ext)
         if nargin > 0
            n = numel(nme);
            obj = repelem(obj,1,n);
            for k = 1:n
               obj(k).Name = nme{k};
               obj(k).PhoneX = ext(k);
            end
         else
            obj.Name = '';
         end
      end
   end
end
```

The `ProjectEngineer` class represents one type of team member. This class supports array inputs and returns an array of objects.

```
classdef ProjectEngineer < TeamMembers
   % Inputs: {Name}, [PhoneX], {Rate}
   properties
      Rate
   end
   methods
      function obj = ProjectEngineer(varargin)
         obj = obj@TeamMembers(varargin{1:2});
         for k = 1:numel(varargin{1})
            obj(k).Rate = varargin{3}{k};
         end
      end
   end
end
```

The `ProjectEngineer` class requires a cell array of names, a numeric array of phone extensions, and a cell array of billing rates for each engineer in the team.

```
nm = {'Fred','Nancy','Claudette'};
px = [8112,8113,8114];
rt = {'C2','B1','A2'};
tm = ProjectEngineer(nm,px,rt)
```

```
tm =

  1x3 ProjectEngineer array with properties:

    Rate
    Name
    PhoneX
```

Potential Error

The `TeamMembers` constructor initializes the object array with this statement:

```
obj = repelem(obj,1,n);
```

Because the `obj` argument to `repelem` is a `ProjectEngineer` object, the array returned is of the same class.

Without this statement, the `TeamMembers` constructor would create default objects to fill in array elements in the `for` loop. The resulting heterogeneous array would be of the class of the common superclass (`TeamMembers` in this case). If the superclass returns this heterogeneous array to the subclass constructor, it is a violation of the rule that class constructors must preserve the class of the returned object.

MATLAB issues this error:

```
When constructing an instance of class 'ProjectEngineer', the constructor must
preserve the class of the returned object.

Error in ProjectEngineer (line 8)
        obj = obj@TeamMembers(varargin{1:2});
```

See Also

More About

- “Designing Heterogeneous Class Hierarchies” on page 10-22

Events — Sending and Responding to Messages

Overview Events and Listeners

In this section...

Why Use Events and Listeners

Events are notices that objects broadcast in response to something that happens, such as a property value changing or a user interaction with an application program. Listeners execute functions when notified that the event of interest occurs. Use events to communicate changes to objects. Listeners respond by executing the callback function.

For more information, see “Event and Listener Concepts” on page 11-12.

Events and Listeners Basics

When using events and listeners:

- Only `handle` classes can define events and listeners.
- Define event names in the `events` block of a class definition (“Events and Listeners Syntax” on page 11-18).
- Use event attributes to specify access to the event (“Event Attributes” on page 11-16).
- Call the handle `notify` method to trigger the event. The event notification broadcasts the named event to all listeners registered for this event.
- Use the handle `addlistener` method to couple a listener to the event source object. MATLAB destroys the listener when the source of the event is destroyed.
- Use the handle `listener` method to create listeners that are not coupled to the lifecycle of the event source object. This approach is useful when the event source and the listeners are defined in different components that you want to be able to add, remove, or modify independently. Your application code controls the listener object lifecycle.
- Listener callback functions must define at least two input arguments — the event source object handle and the event data (See “Listener Callback Syntax” on page 11-24 for more information).
- Modify the data passed to each listener callback by subclassing the `event.EventData` class.

Predefined Events

MATLAB defines events for listening to property sets and queries. For more information, see “Listen for Changes to Property Values” on page 11-32.

All handle objects define an event named `ObjectBeingDestroyed`. MATLAB triggers this event before calling the class destructor.

Event Syntax

Define an event name in the `events` code block:

```
classdef ClassName < handle
   events
      EventName
   end
end
```

For example, `MyClass` defines an event named `StateChange`:

```
classdef MyClass < handle
   events
      StateChange
   end
end
```

Trigger an event using the `handle` class `notify` method:

```
classdef ClassName < handle
   events
      EventName
   end

   methods
      function anyMethod(obj)
         notify(obj,'EventName');
      end
   end
end
```

Any function or method can trigger the event for a specific instance of the class defining the event. For example, the `triggerEvent` method calls `notify` to trigger the `StateChange` event:

```
classdef MyClass < handle
   events
      StateChange
   end
   methods
      function triggerEvent(obj)
         notify(obj,'StateChange')
      end
   end
end
```

Trigger the `StateChange` event with the `triggerEvent` method:

```
obj = MyClass;
obj.triggerEvent
```

For more information, see “Events and Listeners Syntax” on page 11-18.

Create Listener

Define a listener using the `handle` class `addlistener` or `listener` method. Pass a function handle for the listener callback function using one of these syntaxes:

- `addlistener(SourceOfEvent,'EventName',@functionName)` — for an ordinary function.
- `addlistener(SourceOfEvent,'EventName',@Obj.methodName)` — for a method of `Obj`.

- `addlistener(SourceOfEvent,'EventName',@ClassName.methodName)` — for a static method of the class `ClassName`.

```
ListenerObject = addlistener(SourceOfEvent,'EventName',@listenerCallback);
```

`addlistener` returns the listener object. The input arguments are:

- *`SourceOfEvent`* — An object of the class that defines the event. The event is triggered on this object.
- *`EventName`* — The name of the event defined in the class `events` code block.
- `@`*`listenerCallback`* — a function handle referencing the function that executes in response to the event.

For example, create a listener object for the `StateChange` event:

```
function lh = createListener(src)
   lh = addlistener(src,'StateChange',@handleStateChange)
end
```

Define the callback function for the listener. The callback function must accept as the first two arguments the event source object and an event data object: Use the event source argument to access the object that triggered the event. Find information about the event using the event data object.

```
function handleStateChange(src,eventData)
   % src - handle to object that triggered the event
   % eventData - event.EventData object containing
   %              information about the event.
   ...
end
```

For more information, see “Listener Callback Syntax” on page 11-24.

See Also

`event.EventData` | `handle`

Related Examples

- “Listener Lifecycle” on page 11-23
- “Implement Property Set Listener” on page 11-10

Define Custom Event Data

In this section...

Class Event Data Requirements

Suppose that you want to create a listener callback function that has access to specific information when the event occurs. This example shows how by creating custom event data.

Events provide information to listener callback functions by passing an event data argument to the specified function. By default, MATLAB passes an `event.EventData` object to the listener callback. This object has two properties:

- `EventName` — Name of the event triggered by this object.
- `Source` — Handle of the object triggering the event.

Provide additional information to the listener callback by subclassing the `event.EventData` class.

- Define properties in the subclass to contain the additional data.
- Define a constructor that accepts the additional data as arguments.
- Set the `ConstructOnLoad` class attribute.
- Use the subclass constructor as an argument to the `notify` method to trigger the event.

Define and Trigger Event

The `SimpleEventClass` defines a property set method (see “Property Get and Set Methods” on page 8-38) from which it triggers an event if the property is set to a value exceeding a certain limit. The property set method performs these operations:

- Saves the original property value
- Sets the property to the specified value
- If the specified value is greater than 10, the set method triggers an `Overflow` event
- Passes the original property value, and other event data, in a `SpecialEventDataClass` object to the `notify` method.

```
classdef SimpleEventClass < handle
   properties
      Prop1 = 0
   end
   events
      Overflow
   end
   methods
      function set.Prop1(obj,value)
         orgvalue = obj.Prop1;
```

```
            obj.Prop1 = value;
            if (obj.Prop1 > 10)
               % Trigger the event using custom event data
               notify(obj,'Overflow',SpecialEventDataClass(orgvalue));
            end
         end
      end
end
```

Define Event Data

Event data is always contained in an `event.EventData` object. The `SpecialEventDataClass` adds the original property value to the event data by subclassing `event.EventData`:

```
classdef (ConstructOnLoad) SpecialEventDataClass < event.EventData
   properties
      OrgValue = 0
   end
   methods
      function eventData = SpecialEventDataClass(value)
         eventData.OrgValue = value;
      end
   end
end
```

Create Listener for Overflow Event

To listen for the `Overflow` event, attach a listener to an instance of the `SimpleEventClass` class. Use the `addlistener` method to create the listener. Also, you must define a callback function for the listener to execute when the event is triggered.

The function `setupSEC` instantiates the `SimpleEventClass` class and adds a listener to the object. In this example, the listener callback function displays information that is contained in the `eventData` argument (which is a `SpecialEventDataClass` object).

```
function sec = setupSEC
   sec = SimpleEventClass;
   addlistener(sec,'Overflow',@overflowHandler)
   function overflowHandler(eventSrc,eventData)
      disp('The value of Prop1 is overflowing!')
      disp(['Its value was: ' num2str(eventData.OrgValue)])
      disp(['Its current value is: ' num2str(eventSrc.Prop1)])
   end
end
```

Create the `SimpleEventClass` object and add the listener:

```
sec = setupSEC;
sec.Prop1 = 5;
sec.Prop1 = 15; % listener triggers callback

The value of Prop1 is overflowing!
Its value was: 5
Its current value is: 15
```

See Also

Related Examples

- “Observe Changes to Property Values” on page 11-8

Observe Changes to Property Values

This example shows how to listen for changes to a property value. This example uses:

- `PostSet` event predefined by MATLAB
- `SetObservable` property attribute to enable triggering the property `PostSet` event.
- `addlistener` handle class method to create the listener

```
classdef PropLis < handle
   % Define a property that is SetObservable
   properties (SetObservable)
      ObservedProp = 1
   end
   methods
      function attachListener(obj)
         %Attach a listener to a PropListener object
         addlistener(obj,'ObservedProp','PostSet',@PropLis.propChange);
      end
   end
   methods (Static)
      function propChange(metaProp,eventData)
         % Callback for PostSet event
         % Inputs: meta.property object, event.PropertyEvent
         h = eventData.AffectedObject;
         propName = metaProp.Name;
         disp(['The ',propName,' property has changed.'])
         disp(['The new value is: ',num2str(h.ObservedProp)])
         disp(['Its default value is: ',num2str(metaProp.DefaultValue)])
      end
   end
end
```

The `PropLis` class uses an ordinary method (`attachListener`) to add the listener for the `ObservedProp` property. If the `PropLis` class defines a constructor, the constructor can contain the call to `addlistener`.

The listener callback is a static method (`propChange`). MATLAB passes two arguments when calling this function:

- `metaProp` — a `meta.property` object for `ObservedProp`
- `eventData` — an `event.PropertyEvent` object contain event-specific data.

These arguments provide information about the property and the event.

Use the `PropLis` class by creating an instance and calling its `attachListener` method:

```
plObj = PropLis;
plObj.ObservedProp

ans =

     1

plObj.attachListener
plObj.ObservedProp = 2;
```

```
The ObservedProp property has changed.
The new value is: 2
Its default value is: 1
```

See Also

`event.proplistener` | `addlistener` | `listener`

Related Examples

- “Listener Lifecycle” on page 11-23
- “Implement Property Set Listener” on page 11-10

Implement Property Set Listener

This example shows how to define a listener for a property set event. The listener callback triggers when the value of a specific property changes. The class defined for this example uses a method for a push-button callback and a static method for the listener callback. When the push-button callback changes the value of a property, the listener executes its callback on the `PreSet` event.

This example defines a class (`PushButton`) with these design elements:

- `ResultNumber` – Observable property
- uicontrol `pushbutton` – Push-button object used to generate a new graph when its callback executes
- A listener that responds to a change in the observable property

PushButton Class Design

The `PushButton` class creates `figure`, `uicontrol`, `axes` graphics objects, and a listener object in the class constructor.

The push button's callback is a class method (named `pressed`). When the push button is activated, the following sequence occurs:

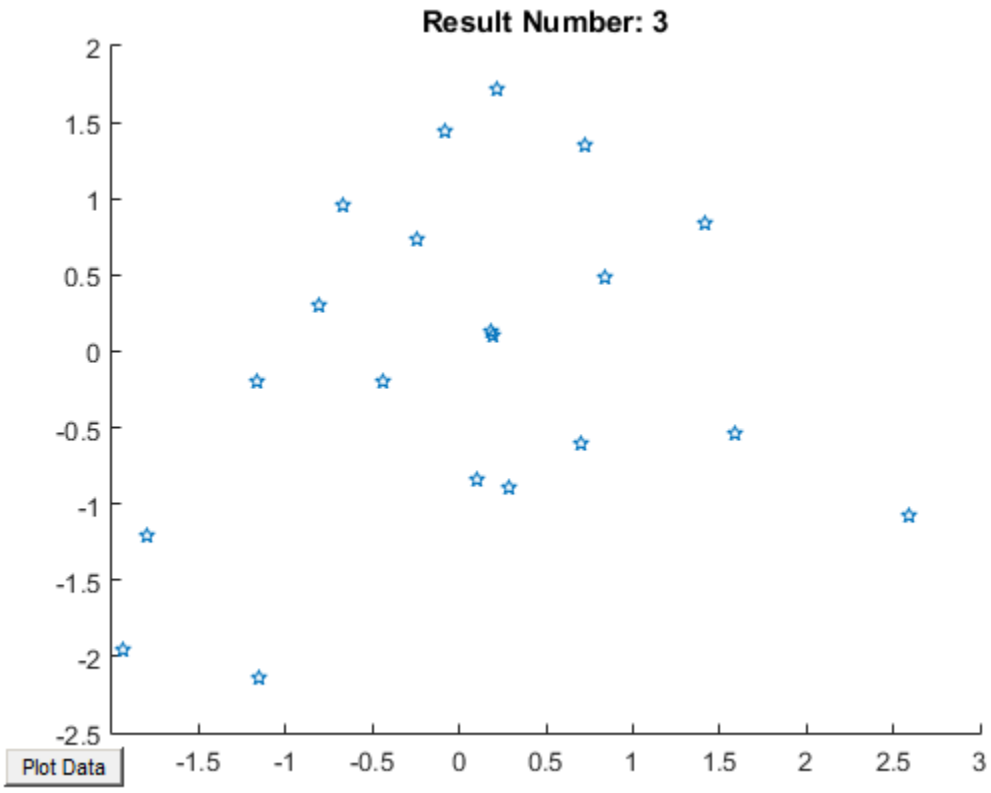
1. MATLAB executes the `pressed` method, which graphs a new set of data and increments the `ResultNumber` property.
2. Attempting to set the value of the `ResultNumber` property triggers the `PreSet` event, which executes the listener callback before setting the property value.
3. The listener callback uses the event data to obtain the handle of the callback object (an instance of the `PushButton` class), which then provides the handle of the axes object that is stored in its `AxHandle` property.
4. The listener callback updates the axes `Title` property, after the callback completes execution, MATLAB sets the `ResultsNumber` property to its new value.

```
classdef PushButton < handle
   properties (SetObservable)
      ResultNumber = 1
   end
   properties
      AxHandle
   end
   methods
      function buttonObj = PushButton
         myFig = figure;
         buttonObj.AxHandle = axes('Parent',myFig);
         uicontrol('Parent',myFig,...
            'Style','pushbutton',...
            'String','Plot Data',...
            'Callback',@(src,evnt)pressed(buttonObj));
         addlistener(buttonObj,'ResultNumber','PreSet',...
            @PushButton.updateTitle);
      end
   end
   methods
      function pressed(obj)
```

```
            scatter(obj.AxHandle,randn(1,20),randn(1,20),'p')
            obj.ResultNumber = obj.ResultNumber + 1;
        end
    end
    methods (Static)
        function updateTitle(~,eventData)
            h = eventData.AffectedObject;
            set(get(h.AxHandle,'Title'),'String',['Result Number: ',...
                num2str(h.ResultNumber)])
        end
    end
end
```

The scatter graph looks similar to this graph after three push-button clicks.

```
buttonObj = PushButton;
```



See Also

Related Examples

- “Listen for Changes to Property Values” on page 11-32

Event and Listener Concepts

In this section...

The Event Model

Events represent changes or actions that occur within objects. For example,

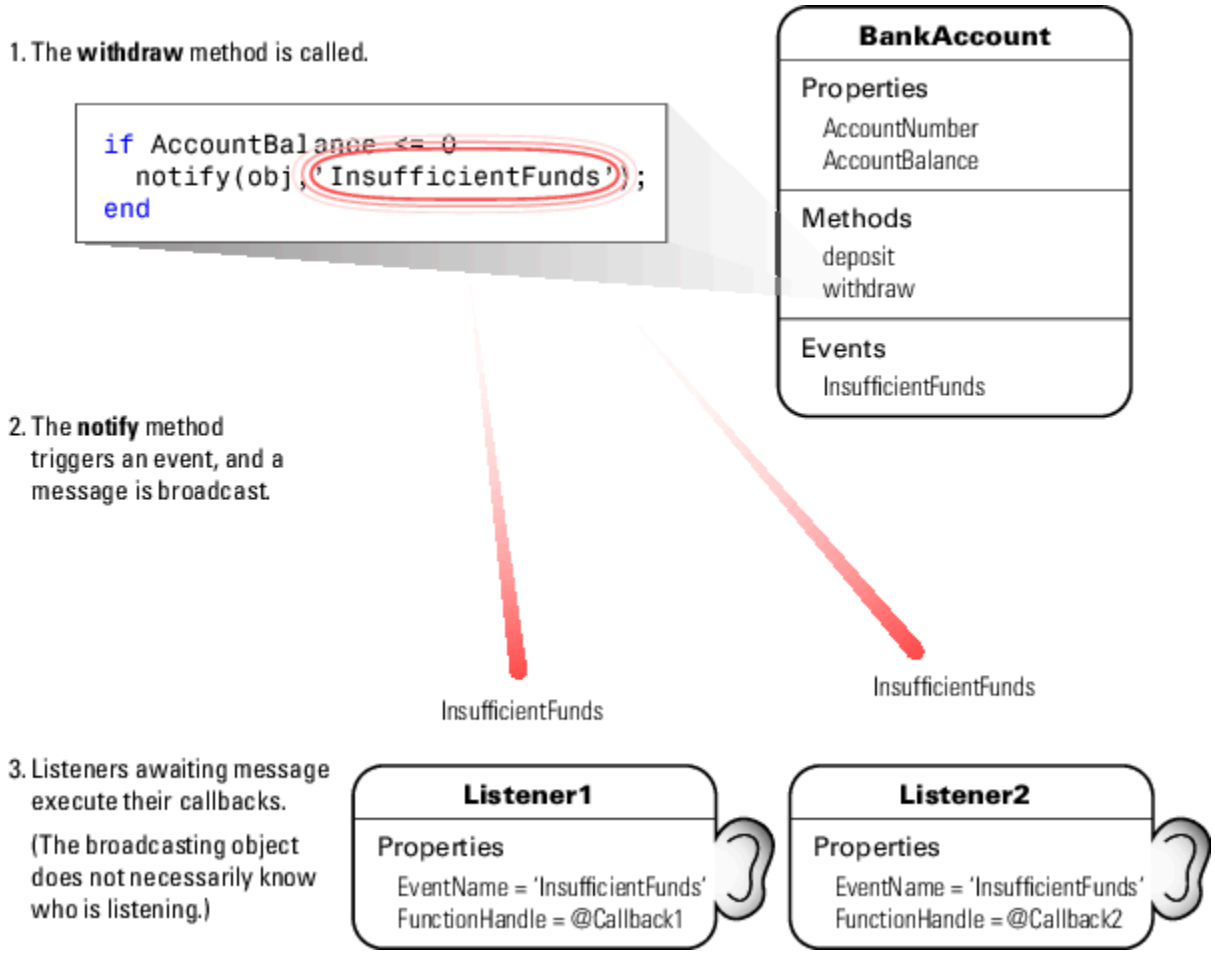
- Modification of class data
- Execution of a method
- Querying or setting a property value
- Destruction of an object

Basically, any activity that you can detect programmatically can generate an event and communicate information to other objects.

MATLAB classes define a process that communicates the occurrence of events to other objects that respond to the events. The event model works this way:

- A handle class declares a name used to represent an event. “Name Events” on page 11-18
- After creating an object of the event-declaring class, attach listener to that object. “Control Listener Lifecycle” on page 11-23
- A call to the handle class `notify` method broadcasts a notice of the event to listeners. The class user determines when to trigger the event. “Trigger Events” on page 11-18
- Listeners execute a callback function when notified that the event has occurred. “Specifying Listener Callbacks” on page 11-24
- You can bind listeners to the lifecycle of the object that defines the event, or limit listeners to the existence and scope of the listener object. “Control Listener Lifecycle” on page 11-23

This diagram illustrates the event model.



Limitations

There are certain limitations to the use of events:

- The event source cannot guarantee that listeners exist when triggering the event.
- A listener cannot prevent other listeners from being notified that the event occurred.
- The order in which listeners execute is not defined.
- Listeners should not modify the event data object passed to the listener callback, because other listeners are passed this same handle object.

Default Event Data

Events provide information to listener callbacks by passing an event data argument to the callback function. By default, MATLAB passes an `event.EventData` object to the listener callback. This object has two properties:

- `EventName` — The event name as defined in the class `event` block
- `Source` — The object that is the source of the event

MATLAB passes the source object to the listener callback in the required event data argument. Use the source object to access any of the object's public properties from within your listener callback function.

Customize Event Data

You can create a subclass of the `event.EventData` class to provide additional information to listener callback functions. The subclass would define properties to contain the additional data and provide a method to construct the derived event data object so it can be passed to the `notify` method.

“Define Event-Specific Data” on page 11-21 provides an example showing how to customize this data.

Events Only in Handle Classes

You can define events only in handle classes. This restriction exists because a value class is visible only in a single MATLAB workspace so no callback or listener can have access to the object that triggered the event. The callback could have access to a copy of the object. However, accessing a copy is not useful because the callback cannot access the current state of the object that triggered the event or effect any changes in that object.

“Comparison of Handle and Value Classes” on page 7-2 provides general information on handle classes.

“Events and Listeners Syntax” on page 11-18 shows the syntax for defining a handle class and events.

Property-Set and Query Events

There are four predefined events related to properties:

- `PreSet` — Triggered just before the property value is set, before calling its set access method
- `PostSet` — Triggered just after the property value is set
- `PreGet` — Triggered just before a property value query is serviced, before calling its get access method
- `PostGet` — Triggered just after returning the property value to the query

These events are predefined and do not need to be listed in the class `events` block.

When a property event occurs, the callback is passed an `event.PropertyEvent` object. This object has three properties:

- `EventName` — The name of the event described by this data object
- `Source` — The source object whose class defines the event described by the data object
- `AffectedObject` — The object whose property is the source for this event (that is, `AffectedObject` contains the object whose property was either accessed or modified).

You can define your own property-change event data by subclassing the `event.EventData` class. The `event.PropertyEvent` class is a sealed subclass of `event.EventData`.

See “Listen for Changes to Property Values” on page 11-32 for a description of the process for creating property listeners.

See “The PostSet Event Listener” on page 11-48 for an example.

See “Property Get and Set Methods” on page 8-38 for information on methods that control access to property values.

Listeners

Listeners encapsulate the response to an event. Listener objects belong to the `event.listener` class, which is a handle class that defines the following properties:

- `Source` — Handle or array of handles of the object that generated the event
- `EventName` — Name of the event
- `Callback` — Function to execute when an enabled listener receives event notification
- `Enabled` — Callback function executes only when `Enabled` is `true`. See “Enable and Disable Listeners” on page 11-50 for an example.
- `Recursive` — Allow listener to trigger the same event that caused execution of the callback.

  `Recursive` is `false` by default. If the callback triggers the event for which it is defined as the callback, the listener cannot execute recursively. Therefore, set `Recursive` to `true` if the callback must trigger its own event. Setting the `Recursive` property to `true` can create a situation where infinite recursion reaches the recursion limit and triggers an error.

“Control Listener Lifecycle” on page 11-23 provides more specific information.

Event Attributes

Specify Event Attributes

The following table lists the attributes you can set for events. To specify a value for an attribute, assign the attribute value on the same line as the event keyword. For example, all the events defined in the following `events` block have protected `ListenAccess` and private `NotifyAccess`.

```
events (ListenAccess = protected, NotifyAccess = private)
   EventName1
   EventName2
end
```

To define other events in the same class definition that have different attribute settings, create another `events` block.

Event Attributes

Attribute Name	Class	Description
`Hidden`	`logical` Default = `false`	If `true`, event does not appear in list of events returned by `events` function (or other event listing functions or viewers).

`ListenAccess`	• enumeration, default = `public`
	• `meta.class` object
	• cell array of `meta.class` objects
		Determines where you can create listeners for the event.
		• `public` — Unrestricted access
		• `protected` — Access from methods in class or subclasses
		• `private` — Access by class methods only (not from subclasses)
		• List classes that have listen access to this event. Specify classes as `meta.class` objects in the form:
		  • A single `meta.class` object
		  • A cell array of `meta.class` objects. An empty cell array, `{}`, is the same as `private` access.
		See “Class Members Access” on page 12-23

`NotifyAccess`	• enumeration, default = `public`
	• `meta.class` object
	• cell array of `meta.class` objects
		Determines where code can trigger the event
		• `public` — Any code can trigger event
		• `protected` — Can trigger event from methods in class or derived classes
		• `private` — Can trigger event by class methods only (not from derived classes)
		• List classes that have notify access to this event. Specify classes as `meta.class` objects in the form:
		  • A single `meta.class` object
		  • A cell array of `meta.class` objects. An empty cell array, `{}`, is the same as `private` access.
		See “Class Members Access” on page 12-23

Framework attributes	Classes that use certain framework base classes have framework-specific attributes. See the documentation for the specific base class you are using for information on these attributes.	

See Also

Related Examples

- “Events and Listeners Syntax” on page 11-18

Events and Listeners Syntax

In this section...

Components to Implement

Implementation of events and listeners involves these components:

- Specification of the name of an event in a handle class — “Name Events” on page 11-18.
- A function or method to trigger the event when the action occurs — “Trigger Events” on page 11-18.
- Listener objects to execute callback functions in response to the triggered event — “Listen to Events” on page 11-19.
- Default or custom event data that the event passes to the callback functions — “Define Event-Specific Data” on page 11-21.

Name Events

Define an event by declaring an event name inside an `events` block. For example, this class creates an event called `ToggledState`:

```
classdef ToggleButton < handle
   properties
      State = false
   end
   events
      ToggledState
   end
end
```

Trigger Events

The `OnStateChange` method calls `notify` to trigger the `ToggledState` event. Pass the handle of the object that is the source of the event and the name of the event to `notify`.

```
classdef ToggleButton < handle
   properties
      State = false
   end
   events
      ToggledState
   end
   methods
      function OnStateChange(obj,newState)
         if newState ~= obj.State
```

```
            obj.State = newState;
            notify(obj,'ToggledState');
        end
    end
  end
end
```

Listen to Events

After the call to `notify` triggers an event, MATLAB broadcasts a message to all listeners that are defined for that event and source object. There are two ways to create listeners: using the handle class `addlistener` or `listener` method.

Use addlistener for Persistent Listeners

If you want the listener to persist beyond the normal variable scope, use `addlistener` to create it. The event source object holds a reference to the listener object. When the event source object is destroyed, MATLAB destroys the listener.

This code defines a listener for the `ToggleState` event:

```
lh = addlistener(obj,'ToggleState',@RespondToToggle.handleEvnt);
```

`addlistener` has these arguments:

- `obj` — The object that is the source of the event
- `ToggleState` — The event name passed as a `char` vector
- `@RespondToToggle.handleEvnt` — A function handle to the callback function (see the following definition “Define Listener” on page 11-20).

Use handle.listener to Decouple Listener and Source

Use the `listener` method to create listeners when you want to manage the lifecycle of the listener and do not want a coupling between the event source and listener object. MATLAB does not destroy listeners created with `listener` when the event source is destroyed. However, your code must keep the listener object handle in scope when creating listeners using `listener`.

The `listener` method requires the same arguments as `addlistener`: the event-naming object, the event name, and a function handle to the callback. `listener` returns the handle to the listener object.

```
lh = listener(obj,'EventName',@callbackFunction)
```

For example, this code uses the `ToggleState` event discussed previously:

```
lh = listener(obj,'ToggleState',@RespondToToggle.handleEvnt)
```

Callback Function

The listener callback function must accept a minimum of two arguments, which MATLAB automatically passes to the callback. Here are the required arguments:

- The source of the event — that is, `obj` in the call to `addlistener` or `event.listener`.
- An `event.EventData` object or a subclass of `event.EventData`, such as the `ToggleEventData` object described in, “Define Event-Specific Data” on page 11-21.

Define the callback function to accept the source object and event data arguments.

```
function callbackFunction(src,evtdata)
   ...
end
```

For more information on callback syntax, see “Listener Callback Syntax” on page 11-24.

Define Listener

The `RespondToToggle` class defines objects that listen for the `ToggleState` event defined in the `ToggleButton` class.

```
classdef RespondToToggle < handle
   methods
      function obj = RespondToToggle(toggle_button_obj)
         addlistener(toggle_button_obj,'ToggledState',@RespondToToggle.handleEvnt);
      end
   end
   methods (Static)
      function handleEvnt(src,~)
         if src.State
            disp('ToggledState is true')
         else
            disp('ToggledState is false')
         end
      end
   end
end
```

The class `RespondToToggle` adds the listener in its constructor. In this case, the class defines the callback (`handleEvnt`) as a static method that accepts the two required arguments:

- `src` — The handle of the object triggering the event (that is, a `ToggleButton` object)
- `evtdata` — An `event.EventData` object

For example, this code creates objects of both classes:

```
tb = ToggleButton;
rtt = RespondToToggle(tb);
```

Whenever you call the `OnStateChange` method of the `ToggleButton` object, `notify` triggers the event. For this example, the callback displays the value of the `State` property:

```
tb.OnStateChange(true)
```

```
ToggledState is true
```

```
tb.OnStateChange(false)
```

```
ToggledState is false
```

Remove Listeners

Remove a listener object by calling `delete` on its handle. For example, if the class `RespondToToggle` saved the listener handle as a property, you could delete the listener.

```
classdef RespondToToggle < handle
   properties
      ListenerHandle % Property for listener handle
   end
   methods
      function obj = RespondToToggle(toggle_button_obj)
```

```
            hl = addlistener(toggle_button_obj,'ToggledState',@RespondToToggle.handleEvnt);
            obj.ListenerHandle = hl; % Save listener handle
        end
    end
    methods (Static)
        function handleEvnt(src,~)
            if src.State
                disp('ToggledState is true')
            else
                disp('ToggledState is false')
            end
        end
    end
end
```

With this code change, you can remove the listener from an instance of the `RespondToToggle` class. For example:

```
tb = ToggleButton;
rtt = RespondToToggle(tb);
```

The object `rtt` is listening for the `ToggleState` event triggered by object `tb`. To remove the listener, call `delete` on the property containing the listener handle.

```
delete(rtt.ListenerHandle)
```

To deactivate a listener temporarily, see “Temporarily Deactivate Listeners” on page 11-23.

Define Event-Specific Data

Suppose that you want to pass the state of the toggle button as a result of the event to the listener callback. You can add more data to the default event data by subclassing the `event.EventData` class and adding a property to contain this information. Then you can pass this object to the `notify` method.

Note To save and load objects that are subclasses of `event.EventData`, such as `ToggleEventData`, enable the `ConstructOnLoad` class attribute for the subclass.

```
classdef (ConstructOnLoad) ToggleEventData < event.EventData
    properties
        NewState
    end

    methods
        function data = ToggleEventData(newState)
            data.NewState = newState;
        end
    end
end
```

The call to `notify` can use the `ToggleEventData` constructor to create the necessary argument.

```
evtdata = ToggleEventData(newState);
notify(obj,'ToggledState',evtdata);
```

See Also

Related Examples

- “Listener Callback Syntax” on page 11-24
- “Listen for Changes to Property Values” on page 11-32
- “Techniques for Using Events and Listeners” on page 11-40

Listener Lifecycle

In this section...

Control Listener Lifecycle

There are two ways to create listeners:

- `addlistener` creates a coupling between the listener and event source object. The listener object persists until you delete it or until the event object is destroyed. When the event source object is destroyed, MATLAB automatically destroys the listener object.
- `listener` constructs listener objects that are not coupled to the lifecycle of the event source object. The listener is active as long as the listener object remains in scope and is not explicitly deleted. Therefore, your application must maintain a reference to the listener object by storing the listener handle. The advantage of uncoupling the listener and event objects is that you can define and destroy each independently.

For more information, see “Events and Listeners Syntax” on page 11-18.

Temporarily Deactivate Listeners

The `addlistener` method returns the listener object so that you can set its properties. For example, you can temporarily disable a listener by setting its `Enabled` property to `false`:

```
ListenerHandle.Enabled = false;
```

To reenable the listener, set `Enabled` to `true`.

```
ListenerHandle.Enabled = true;
```

Permanently Delete Listeners

Calling `delete` on a listener object destroys it and permanently removes the listener:

```
delete(ListenerHandle)
```

See Also

Related Examples

- “Enable and Disable Listeners” on page 11-50

Listener Callback Syntax

In this section...

Specifying Listener Callbacks

Callbacks are functions that execute when the listener receives notification of the event. Pass a function handle referencing the callback function to `addlistener` or `listener` when creating the listener.

All callback functions must accept at least two arguments:

- The handle of the object that is the source of the event
- An `event.EventData` object or an object that is derived from the `event.EventData` class.

Syntax to Reference Callback

For a function: `functionName`

```
lh = addlistener(eventSourceObj,'EventName',@functionName)
```

For an ordinary method called with an object of the class: `obj.methodName`

```
lh = addlistener(eventSourceObj,'EventName',@obj.methodName)
```

For a static method:`ClassName.methodName`

```
lh = addlistener(eventSourceObj,'EventName',@ClassName.methodName)
```

For a function in a package:`PackageName.functionName`

```
lh = addlistener(eventSourceObj,'EventName',@PackageName.functionName)
```

Input Arguments for Callback Function

Define the callback function to accept the required arguments:

```
function callbackFunction(src,evnt)
   ...
end
```

If you do not use the event source and event data arguments, you can define the function to ignore these inputs:

```
function callbackFunction(~,~)
   ...
end
```

For a method:

```
function callbackMethod(obj,src,evnt)
   ...
end
```

Additional Arguments for Callback Function

To pass arguments to your callback in addition to the source and event data arguments passed by MATLAB, use an anonymous function. Anonymous functions can use any variables that are available in the current workspace.

Syntax Using Anonymous Function

Here is the syntax for an ordinary method. The input arguments (`arg1,...argn`) must be defined in the context in which you call `addlistener`.

```
lh = addlistener(src,'EventName',@(src,evnt)obj.callbackMethod(src,evnt,arg1,...argn)
```

Use `varargin` to define the callback function.

```
function callbackMethod(src,evnt,varargin)
   arg1 = varargin{1};
   ...
   argn = varargin{n};
   ...
end
```

For general information on anonymous function, see “Anonymous Functions”.

Using Methods for Callbacks

The `TestAnonyFcn` class shows the use of an anonymous function with an additional argument. The listener callback displays the inputs arguments to show how MATLAB calls the callback method.

```
classdef TestAnonyFcn < handle
   events
      Update
   end
   methods
      function obj = TestAnonyFcn
         t = datestr(now);
         addlistener(obj,'Update',@(src,evnt)obj.evntCb(src,evnt,t));
      end
      function triggerEvnt(obj)
         notify(obj,'Update')
      end
   end
   methods (Access = private)
      function evntCb(~,~,evnt,varargin)
         disp(['Number of inputs: ',num2str(nargin)])
         disp(evnt.EventName)
         disp(varargin{:})
      end
   end
end
```

Create an object and trigger the event by calling the `triggerEvt` method:

```
obj = TestAnonyFcn;
obj.triggerEvnt;
```

```
Number of inputs: 4
Update
01-Jul-2008 17:19:36
```

See Also

Related Examples

- “Callback Execution” on page 11-27
- “Create Function Handle”

Callback Execution

In this section...

When Callbacks Execute

Listeners execute their callback function when notified that the event has occurred. Listeners are passive observers in the sense that errors in the execution of a listener callback do not prevent the execution of other listeners responding to the same event, or execution of the function that triggered the event.

Callback function execution continues until the function completes. If an error occurs in a callback function, execution stops and control returns to the calling function. Then any remaining listener callback functions execute.

Listener Order of Execution

The order in which listeners callback functions execute after the firing of an event is undefined. However, all listener callbacks execute synchronously with the event firing.

The handle class `notify` method calls all listeners before returning execution to the function that called `notify`.

Callbacks That Call notify

Do not modify and reuse or copy and reuse the event data object that you pass to `notify`, which is then passed to the listener callback.

Listener callbacks can call `notify` to trigger events, including the same event that invoked the callback. When a function calls `notify`, MATLAB sets the property values of the event data object that is passed to callback functions. To ensure that these properties have appropriate values for subsequently called callbacks, always create a new event data object if you call `notify` with custom event data.

Manage Callback Errors

If you want to control how your program responds to errors, use a `try/catch` statement in your listener callback function to handle errors.

Invoke Functions from Function Handles

When you create a function handle inside a class method, the context of the method determines the context in which the function executes. This context gives the function access to private and protected methods that are accessible to that class.

For example, the `UpdateEvt` class defines an event named `Update` and a listener for that event. The listener callback is the private method `evtCb`.

```
classdef UpdateEvt < handle
   events
      Update
   end
   methods
      function obj = UpdateEvt
         addlistener(obj,'Update',@evtCb);
      end
   end
   methods (Access = private)
      function obj = evtCb(obj,varargin)
         disp('Updated Event Triggered')
      end
   end
end
```

Private methods are normally accessible only by class methods. However, because the function handle is created in a class method, `notify` can execute the callback from outside of the class:

```
a = UpdateEvt;
a.notify('Update')
```

```
Updated Event Triggered
```

See Also

Related Examples

- “Listener Callback Syntax” on page 11-24

Determine If Event Has Listeners

In this section...

Do Listeners Exist for This Event?

Use the `event.hasListener` function to determine if a specific event has listeners. `event.hasListener` accepts an array of event source objects and an event name as input arguments. It returns an array of logical values indicating if listeners exist for the specified event on each object in the array.

Note When called, `event.hasListener` must have `NotifyAccess` for the event. That is, call `event.hasListener` in a context in which you can call `notify` for the event in question.

Why Test for Listeners

Use `event.hasListener` to avoid sending event notifications when there are no listeners for the event. For example, if creating custom event data consumes significant resources, or if events are triggered repeatedly, use `event.hasListener` to test for listeners before performing these steps.

Coding Patterns

- Conditionalize the creation of event data and the call to `notify` using `event.hasListener`. For an object array `a`, determine if there are listeners before creating event data and triggering the event:

  ```
  if any(event.hasListener(a,'NameOfEvent'))
     evt = MyCustomEventData(...);
     notify(a,'NameOfEvent',evt)
  end
  ```

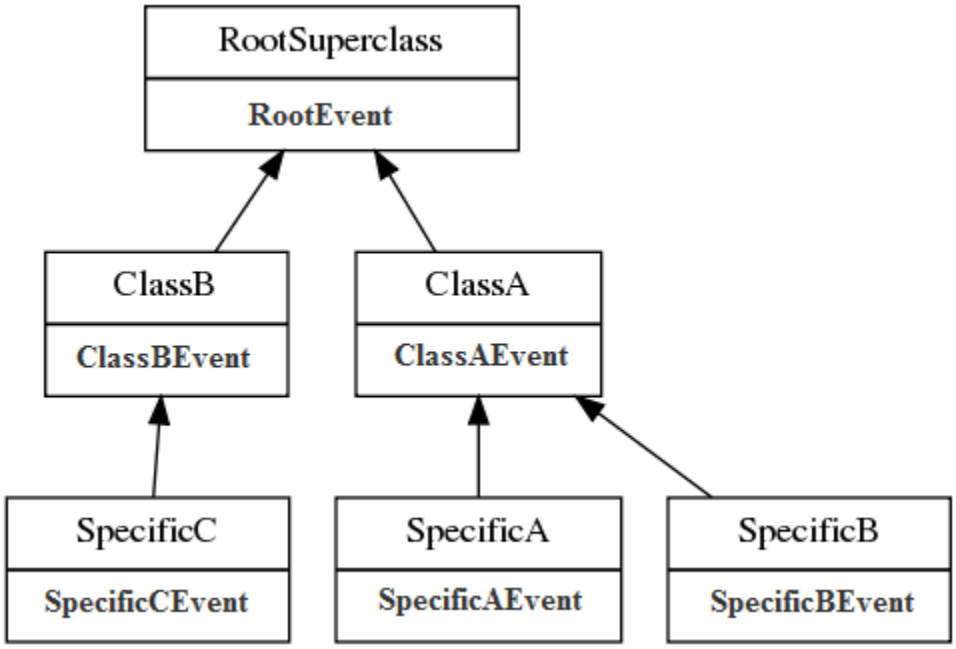
- Trigger events selectively using logical indexing with the values returned by `event.hasListener`. Send event notifications only for array elements that have listeners:

  ```
  ind = event.hasListeners(a,'NameOfEvent');
  notify(a(ind),'NameOfEvent',evt)
  ```

Listeners in Heterogeneous Arrays

If the input object array is heterogeneous, the class of the array must define the specified event. You can query the listeners only for events that all objects in the array define.

For example, in the following diagram, the class of a heterogeneous array formed with objects of classes `SpecificA`, `SpecificB`, and `SpecificC` is `RootSuperclass`. Therefore, `event.hasListener` can find listeners only for the `RootEvent` event, because it is the only event common to all array elements.



Create a heterogeneous array with the three most specific classes:

```
het = [SpecificA,SpecificB,SpecificC];
class(het)

ans

RootSuperclass

events(het)

Events for class RootSuperclass

   RootEvent
```

`event.hasListener` cannot determine if there are listeners for events that are defined by some but not all objects in the array:

```
event.hasListener(het,'ClassAEvent')

Error using event.hasListener
Event 'ClassAEvent' is not defined for class 'RootSuperclass'.
```

Determine if individual objects in the heterogeneous array have listeners defined for their specific events, by indexing into the array:

```
event.hasListener(het(1),'ClassAEvent')
```

For more information about determining the class of heterogeneous arrays, see “Designing Heterogeneous Class Hierarchies” on page 10-22.

See Also

Related Examples

- “Listener Lifecycle” on page 11-23

Listen for Changes to Property Values

In this section...

Create Property Listeners

For handle classes, you can define listeners for the predeclared property events (named: `PreSet`, `PostSet`, `PreGet`, and `PostGet`). To create listeners for those named events:

- Specify the `SetObservable` and/or `GetObservable` property attributes.
- Define callback functions
- Create the property listener by including the name of the property and the event in the call to `addlistener` or `listener`.
- If necessary, subclass `event.data` to create a specialized event data object to pass to the callback function.
- Prevent execution of the callback if the new value is the same as the current value (see “Assignment When Property Value Is Unchanged” on page 11-35).

Set Property Attributes to Enable Property Events

In the properties block, enable the `SetObservable` attribute. You can define `PreSet` and `PostSet` listeners for the properties defined in this block:

```
properties (SetObservable)
   PropOne
   PropTwo
end
```

Define Callback Function for Property Event

The listener executes the callback function when MATLAB triggers the property event. Define the callback function to have two specific arguments, which are passed to the function automatically when called by the listener:

- Event source — a `meta.property` object describing the object that is the source of the property event
- Event data — a `event.PropertyEvent` object containing information about the event

You can pass additional arguments if necessary. It is often simple to define this method as `Static` because these two arguments contain most necessary information in their properties.

For example, suppose the `handlePropEvents` function is a static method of the class creating listeners for two properties of an object of another class:

```
            methods (Static)
   function handlePropEvents(src,evnt)
      switch src.Name
         case 'PropOne'
            % PropOne has triggered an event
         case 'PropTwo'
```

```
            % PropTwo has triggered an event
        end
    end
end
```

Another possibility is to use the `event.PropertyEvent` object's `EventName` property in the `switch` statement to key off the event name (`PreSet` or `PostSet` in this case).

“Class Metadata” on page 16-2 provides more information about the `meta.property` class.

Add Listener to Property

The `addlistener` handle class method enables you to attach a listener to a property without storing the listener object as a persistent variable. For a property event, use the four-argument version of `addlistener`.

Here is a call to `addlistener`:

```
addlistener(EventObject,'PropOne','PostSet',@ClassName.handlePropertyEvents);
```

The arguments are:

- `EventObject` — handle of the object generating the event
- `PropOne` — name of the property to which you want to listen
- `PostSet` — name of the event for which you want to listen
- `@ClassName.handlePropertyEvents` — function handle referencing a static method, which requires the use of the class name

If your listener callback is an ordinary method and not a static method, the syntax is:

```
addlistener(EventObject,'PropOne','PostSet',@obj.handlePropertyEvents);
```

where *`obj`* is the handle of the object defining the callback method.

If the listener callback is a function that is not a class method, you pass a function handle to that function. Suppose that the callback function is a package function:

```
addlistener(EventObject,'PropOne','PostSet',@package.handlePropertyEvents);
```

For more information on passing functions as arguments, see “Create Function Handle”.

Property Event and Listener Classes

The following two classes show how to create `PostSet` property listeners for two properties — `PropOne` and `PropTwo`.

Class Generating the Event

The `PropEvent` class enables property `PreSet` and `PostSet` event triggering by specifying the `SetObservable` property attribute. These properties also enable the `AbortSet` attribute, which prevents the triggering of the property events if the properties are set to a value that is the same as their current value (see “Assignment When Property Value Is Unchanged” on page 11-35).

```
classdef PropEvent < handle
   properties (SetObservable, AbortSet)
      PropOne
      PropTwo
```

```
    end
    methods
       function obj = PropEvent(p1,p2)
          if nargin > 0
             obj.PropOne = p1;
             obj.PropTwo = p2;
          end
       end
    end
end
```

Class Defining the Listeners

The `PropListener` class defines two listeners:

- Property `PropOne` `PostSet` event
- Property `PropTwo` `PostSet` event

You can define listeners for other events or other properties using a similar approach. It is not necessary to use the same callback function for each listener. See the `meta.property` and `event.PropertyEvent` reference pages for more on the information contained in the arguments passed to the listener callback function.

```
classdef PropListener < handle
   % Define property listeners
   methods
      function obj = PropListener(evtobj)
         if nargin > 0
            addlistener(evtobj,'PropOne','PostSet',@PropListener.handlePropEvents);
            addlistener(evtobj,'PropTwo','PostSet',@PropListener.handlePropEvents);
         end
      end
   end
   methods (Static)
      function handlePropEvents(src,evnt)
         switch src.Name
            case 'PropOne'
               sprintf('PropOne is %s\n',num2str(evnt.AffectedObject.PropOne))
            case 'PropTwo'
               sprintf('PropTwo is %s\n',num2str(evnt.AffectedObject.PropTwo))
         end
      end
   end
end
```

See Also

Related Examples

- “Assignment When Property Value Is Unchanged” on page 11-35

Assignment When Property Value Is Unchanged

In this section...

AbortSet When Value Does Not Change

When you set a property value, MATLAB triggers the property `PreSet` and `PostSet` events, invokes the property set method (if one is defined), and sets the property value. These actions occur even when the current value of the property is the same as the new value.

You can prevent these actions by setting the property's `AbortSet` attribute to `true`. When `AbortSet` is enabled, MATLAB compares the current property value to the new value being assigned to the property. If the new value is the same as the current value, MATLAB does not:

- Set the property value.
- Trigger the `PreSet` and `PostSet` events.
- Call the property set method, if one exists.

To compare values, MATLAB must get the current value of the property. Getting the current value causes the property get method (`get.`*`Property`*) to execute, if one exists. Any errors that occur when calling the property get method are visible to the user, even if MATLAB does not change the current value.

Note The `AbortSet` attribute only works with properties defined in handle classes. It cannot be used with value classes.

How MATLAB Compares Values

MATLAB uses the `isequal` function to determine if the current value of the property is the same as the new value. To determine if specific values evaluate as equal when using the `AbortSet` attribute, see the `isequal` function documentation or any `isequal` method overloaded for the class of the property value.

When to Use AbortSet

Use of the `AbortSet` attribute does incur some overhead in the comparison of the current and new property values. Using the `AbortSet` attribute can slow all property assignments because the current and assigned value are always compared before the assignment is made. The `AbortSet` attribute is most useful when:

- You want to prevent notification of the `PreSet` and `PostSet` events and execution of the listener callbacks when the property value does not change.

- The cost of setting a property value is greater than the cost of comparing the current property value with the value being assigned, and you are willing to incur the comparison cost with all assignments to the property.

Implement AbortSet

The following example shows how the `AbortSet` attribute works. The `AbortTheSet` class defines a property, `PropOne`, that has listeners for the `PreGet`, `PreSet`, `PostGet`, and `PostSet` events and enables the `AbortSet` attribute.

Note To use this class, save the `AbortTheSet` class in a file with the same name in a folder on your MATLAB path.

```
classdef AbortTheSet < handle
    properties (SetObservable, GetObservable, AbortSet)
        PropOne = 7
    end
    methods
        function obj = AbortTheSet
            addlistener(obj,'PropOne','PreGet',@obj.getPrePropEvt);
            addlistener(obj,'PropOne','PreSet',@obj.setPrePropEvt);
            addlistener(obj,'PropOne','PostGet',@obj.getPostPropEvt);
            addlistener(obj,'PropOne','PostSet',@obj.setPostPropEvt);
        end
        function propval = get.PropOne(obj)
            disp('get.PropOne called')
            propval = obj.PropOne;
        end
        function set.PropOne(obj,val)
            disp('set.PropOne called')
            obj.PropOne = val;
        end
        function getPrePropEvt(obj,src,evnt)
            disp ('Pre-get event triggered')
            % ...
        end
        function setPrePropEvt(obj,src,evnt)
            disp ('Pre-set event triggered')
            % ...
        end
        function getPostPropEvt(obj,src,evnt)
            disp ('Post-get event triggered')
            % ...
        end
        function setPostPropEvt(obj,src,evnt)
            disp ('Post-set event triggered')
            % ...
        end
        function disp(obj)
            % Overload disp to avoid accessing property
            disp (class(obj))
        end
    end
end
```

The class specifies an initial value of 7 for the `PropOne` property. Therefore, if you create an object and assign the property value of 7, there is no need to trigger the `PreSet` event. However, the `getPropOne` method is called to get the current value of the property to compare to the assigned vale.

```
obj = AbortTheSet;
obj.PropOne = 7;
```

```
get.PropOne called
```

If you specify a value other than 7, then MATLAB performs these steps:

- Gets the current property value
- Triggers the `PreSet` event
- Sets the property to the assigned value
- Triggers the `PostSet` event

```
obj = AbortTheSet;
obj.PropOne = 9;
```

```
get.PropOne called
Pre-set event triggered
set.PropOne called
Post-set event triggered
```

If you query the property value, the `PreGet` and `PostGet` events are triggered.

```
obj.PropOne
```

```
Pre-get event triggered
get.PropOne called
Post-get event triggered

ans =

     9
```

Using AbortSet with Property Validation

When classes use property validation and `AbortSet` in a property definition, MATLAB evaluates the property validation before comparing the current value to the value being assigned. For example, revise the `AbortTheSet` class to add a size restriction of 1-by-3 to the `PropOne` property.

```
classdef AbortTheSet < handle
    properties (SetObservable, GetObservable, AbortSet)
        % Restrict size to 1-by-3
        % ***********************
        PropOne (1,3) = [7 7 7]
        % ***********************
    end
    methods
        function obj = AbortTheSet
            addlistener(obj,'PropOne','PreGet',@obj.getPrePropEvt);
            addlistener(obj,'PropOne','PreSet',@obj.setPrePropEvt);
            addlistener(obj,'PropOne','PostGet',@obj.getPostPropEvt);
            addlistener(obj,'PropOne','PostSet',@obj.setPostPropEvt);
```

```
        end
        function propval = get.PropOne(obj)
            disp('get.PropOne called')
            propval = obj.PropOne;
        end
        function set.PropOne(obj,val)
            disp('set.PropOne called')
            obj.PropOne = val;
        end
        function getPrePropEvt(obj,src,evnt)
            disp ('Pre-get event triggered')
            % ...
        end
        function setPrePropEvt(obj,src,evnt)
            disp ('Pre-set event triggered')
            % ...
        end
        function getPostPropEvt(obj,src,evnt)
            disp ('Post-get event triggered')
            % ...
        end
        function setPostPropEvt(obj,src,evnt)
            disp ('Post-set event triggered')
            % ...
        end
        function disp(obj)
            % Overload disp to avoid accessing property
            disp (class(obj))
        end
    end
end
```

Because MATLAB applies scalar expansion to satisfy the size restriction, the following assignment does not trigger the `PreSet` or `PostSet` events.

```
obj = AbortTheSet;
obj.PropOne = 7;

get.PropOne called

obj.PropOne

Pre-get event triggered
get.PropOne called
Post-get event triggered

ans =

     7     7     7
```

For information on property validation, see “Validate Property Values” on page 8-18.

See Also

Related Examples

- “Property Get and Set Methods” on page 8-38

- “Determine If Event Has Listeners” on page 11-29

Techniques for Using Events and Listeners

In this section...

Example Overview

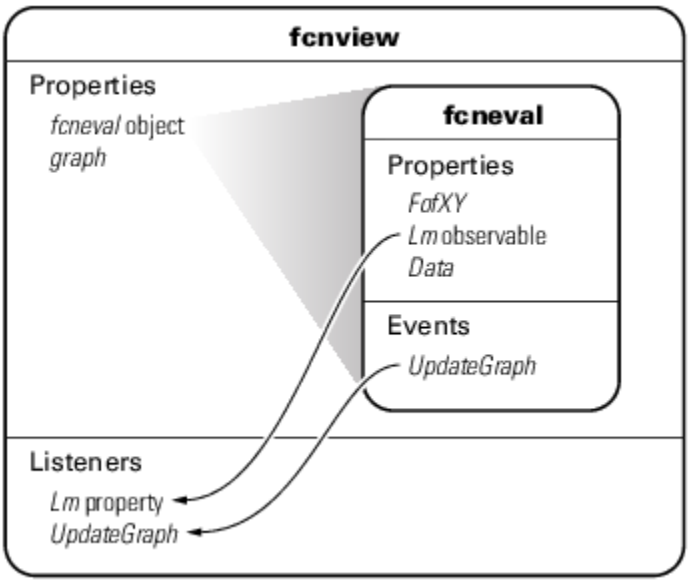
This example defines two classes:

- `fcneval` — The function evaluator class contains a MATLAB expression and evaluates this expression over a specified range
- `fcnview` — The function viewer class contains a `fcneval` object and displays surface graphs of the evaluated expression using the data contained in `fcneval`.

This class defines two events:

- A class-defined event that occurs when a new value is specified for the MATLAB function
- A property event that occurs when the property containing the limits is changed

The following diagram shows the relationship between the two objects. The `fcnview` object contains a `fcneval` object and creates graphs from the data it contains. `fcnview` creates listeners to change the graphs if any of the data in the `fcneval` object change.



Techniques Demonstrated in This Example

- Naming an event in the class definition
- Triggering an event by calling `notify`
- Enabling a property event via the `SetObservable` attribute
- Creating listeners for class-defined events and property `PostSet` events
- Defining listener callback functions that accept additional arguments
- Enabling and disabling listeners

Summary of fcneval Class

The `fcneval` class evaluates a MATLAB expression over a specified range of two variables. The `fcneval` is the source of the data that objects of the `fcnview` class graph as a surface. `fcneval` is the source of the events used in this example. For a listing of the class definition, see “@fcneval/ fcneval.m Class Code” on page 11-51

Property	Value	Purpose
`FofXY`	function handle	MATLAB expression (function of two variables).
`Lm`	two-element vector	Limits over which function is evaluated in both variables. `SetObservable` attribute set to `true` to enable property event listeners.
`Data`	structure with x, y, and z matrices	Data resulting from evaluating the function. Used for surface graph. `Dependent` attribute set to `true`, which means the `get.Data` method is called to determine property value when queried and no data is stored.

Event	When Triggered
UpdateGraph	FofXY property set function (set.FofXY) calls the notify method when a new value is specified for the MATLAB expression on an object of this class.

Method	Purpose
fcneval	Class constructor. Inputs are function handle and two-element vector specifying the limits over which to evaluate the function.
set.FofXY	FofXY property set function. Called whenever property value is set, including during object construction.
set.Lm	Lm property set function. Used to test for valid limits.
get.Data	Data property get function. This method calculates the values for the Data property whenever that data is queried (by class members or externally).
grid	A static method (Static attribute set to true) used in the calculation of the data.

Summary of fcnview Class

Objects of the fcnview class contain fcneval objects as the source of data for the four surface graphs created in a function view. fcnview creates the listeners and callback functions that respond to changes in the data contained in fcneval objects. For a listing of the class definition, see “@fcnview/fcnview.m Class Code” on page 11-52

Property	Value	Purpose
FcnObject	fcneval object	This object contains the data that is used to create the function graphs.
HAxes	axes handle	Each instance of a fcnview object stores the handle of the axes containing its subplot.
HLUpdateGraph	event.listener object for UpdateGraph event	Setting the event.listener object's Enabled property to true enables the listener; false disables listener.
HLLm	event.listener object for Lm property event	Setting the event.listener object's Enabled property to true enables the listener, false disables listener.
HEnableCm	uimenu handle	Item on context menu used to enable listeners (used to handle checked behavior)
HDisableCm	uimenu handle	Item on context menu used to disable listeners (used to manage checked behavior)
HSurface	surface handle	Used by event callbacks to update surface data.

Method	Purpose
fcnview	Class constructor. Input is fcneval object.
createLisn	Calls addlistener to create listeners for UpdateGraph and Lm property PostSet listeners.

Method	Purpose
`lims`	Sets axes limits to current value of `fcneval` object's `Lm` property. Used by event handlers.
`updateSurfaceData`	Updates the surface data without creating a new object. Used by event handlers.
`listenUpdateGraph`	Callback for `UpdateGraph` event.
`listenLm`	Callback for `Lm` property `PostSet` event
`delete`	Delete method for `fcnview` class.
`createViews`	Static method that creates an instance of the `fcnview` class for each subplot, defines the context menus that enable/disable listeners, and creates the subplots

Methods Inherited from Handle Class

Both the `fcneval` and `fcnview` classes inherit methods from the `handle` class. The following table lists only those inherited methods used in this example.

“Handle Class Methods” on page 7-11 provides a complete list of methods that are inherited when you subclass the `handle` class.

Methods Inherited from Handle Class	Purpose
`addlistener`	Register a listener for a specific event and attach listener to event-defining object.
`notify`	Trigger an event and notify all registered listeners.

Using the fcneval and fcnview Classes

This section explains how to use the classes.

- Create an instance of the `fcneval` class to contain the MATLAB expression of a function of two variables and the range over which you want to evaluate this function
- Use the `fcnview` class static function `createViews` to visualize the function
- Change the MATLAB expression or the limits contained by the `fcneval` object and all the `fcnview` objects respond to the events generated.

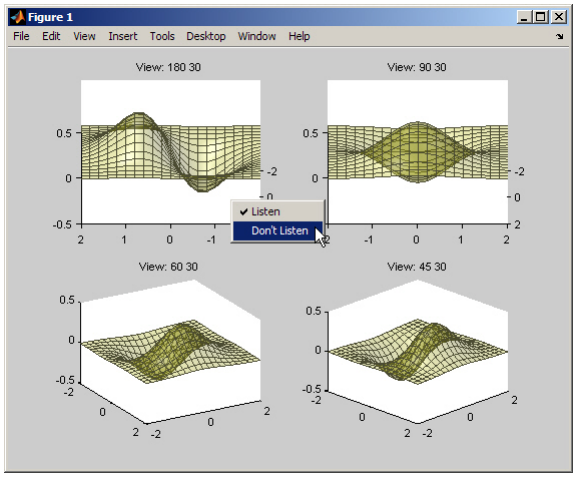
You create a `fcneval` object by calling its constructor with two arguments—an anonymous function and a two-element, monotonically increasing vector. For example:

```
feobject = fcneval(@(x,y) x.*exp(-x.^2-y.^2),[-2 2]);
```

Use the `createViews` static method to create the graphs of the function. Use the class name to call a static function:

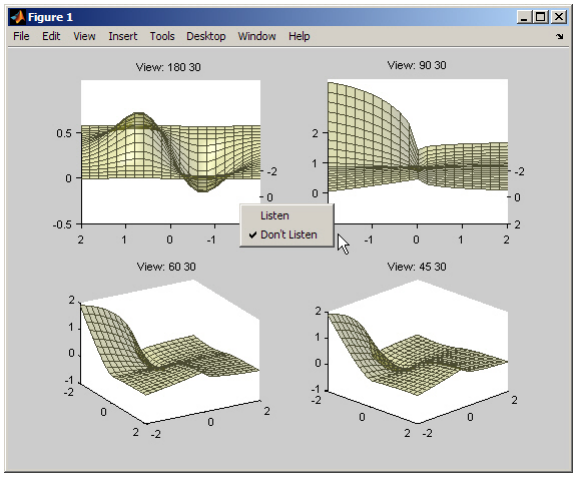
```
fcnview.createViews(feobject);
```

The `createView` method generates four views of the function contained in the `fcneval` object.



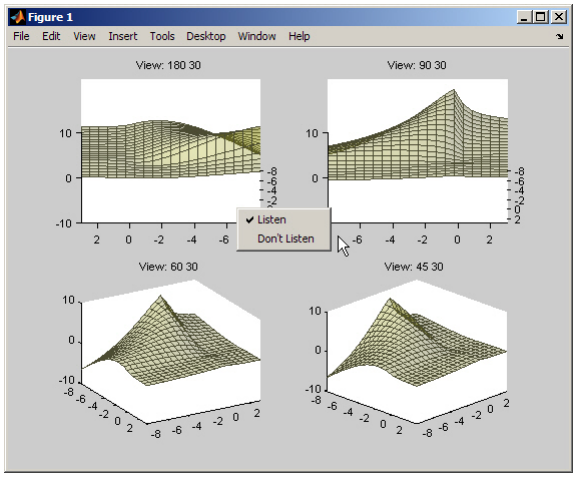
Each subplot defines a context menu that can enable and disable the listeners associated with that graph. For example, if you disable the listeners on subplot 221 (upper left) and change the MATLAB expression contained by the `fcneval` object, only the remaining three subplots update when the `UpdateGraph` event is triggered:

```
feobject.FofXY = @(x,y) x.*exp(-x.^.5-y.^.5);
```



Similarly, if you change the limits by assigning a value to the `feobject.Lm` property, the `feobject` triggers a `PostSet` property event and the listener callbacks update the graph.

```
feobject.Lm = [-8 3];
```



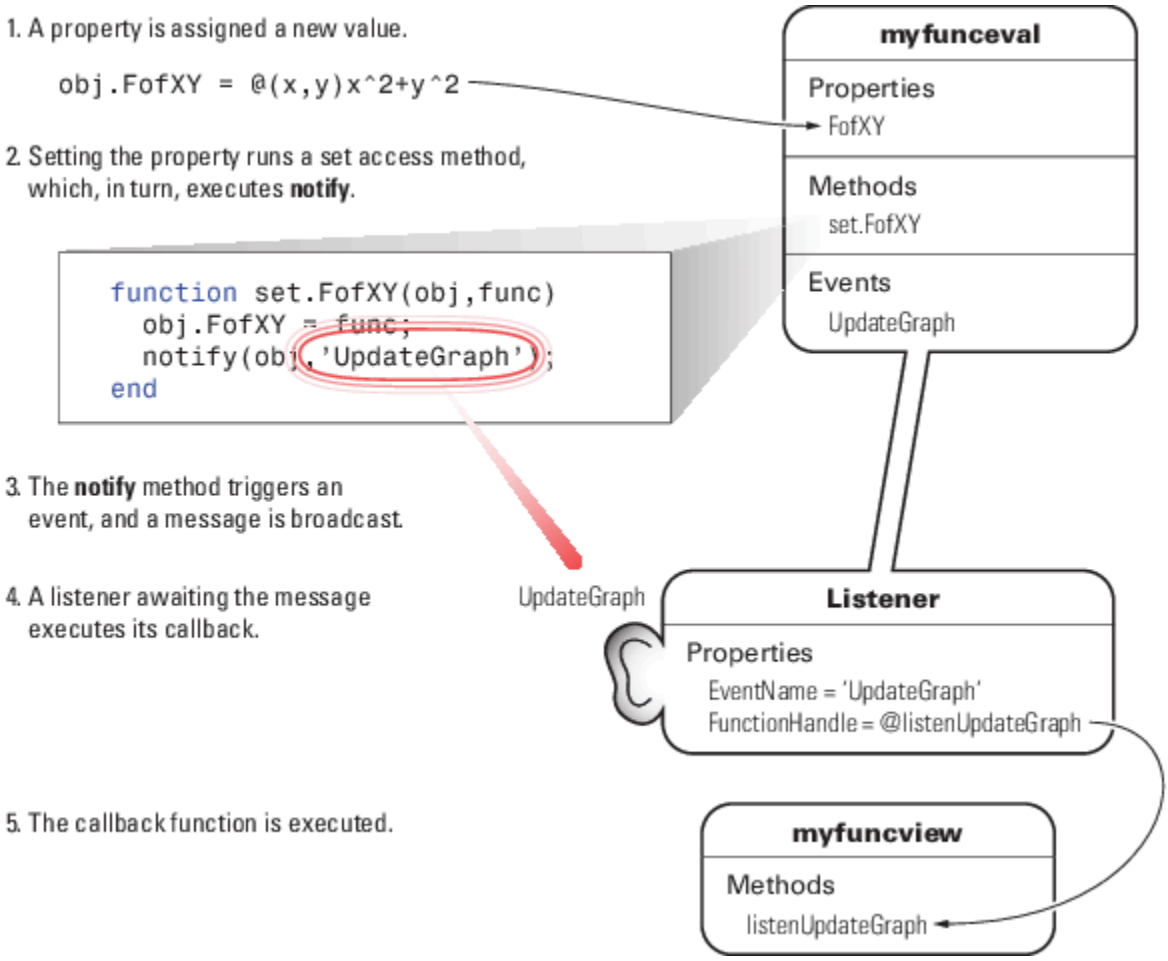
In this figure, the listeners are reenabled via the context menu for subplot 221. Because the listener callback for the property `PostSet` event also updates the surface data, all views are now synchronized

Implement UpdateGraph Event and Listener

The `UpdateGraph` event occurs when the MATLAB representation of the mathematical function contained in the `fcneval` object is changed. The `fcnview` objects that contain the surface graphs are listening for this event, so they can update the graphs to represent the new function.

Define and Trigger UpdateGraph Event

The `UpdateGraph` event is a class-defined event. The `fcneval` class names the event and calls `notify` when the event occurs.



The `fcnview` class defines a listener for this event. When `fcneval` triggers the event, the `fcnview` listener executes a callback function that performs the follow actions:

- Determines if the handle of the surface object stored by the `fcnview` object is still valid (that is, does the object still exist)
- Updates the surface `XData`, `YData`, and `ZData` by querying the `fcneval` object's `Data` property.

The `fcneval` class defines an event name in an `event` block:

```
events
   UpdateGraph
end
```

Determine When to Trigger Event

The `fcneval` class defines a property set method for the `FofXY` property. `FofXY` is the property that stores the MATLAB expression for the mathematical function. This expression must be a valid MATLAB expression for a function of two variables.

The `set.FofXY` method:

- Determines the suitability of the expression
- If the expression is suitable:
  - Assigns the expression to the `FofXY` property
  - Triggers the `UpdateGraph` event

If `fcneval.isSuitable` does not return an `MException` object, the `set.FofXY` method assigns the value to the property and triggers the `UpdateGraph` event.

```
function set.FofXY(obj,func)
% Determine if function is suitable to create a surface
   me = fcneval.isSuitable(func);
   if ~isempty(me)
      throw(me)
   end
% Assign property value
   obj.FofXY = func;
% Trigger UpdateGraph event
   notify(obj,'UpdateGraph');
end
```

Determine Suitability of Expression

The `set.FofXY` method calls a static method (`fcneval.isSuitable`) to determine the suitability of the specified expression. `fcneval.isSuitable` returns an `MException` object if it determines that the expression is unsuitable. `fcneval.isSuitable` calls the `MException` constructor directly to create more useful error messages for the user.

`set.FofXY` issues the exception using the `throw` method. Issuing the exception terminates execution of `set.FofXY` and prevents the method from making an assignment to the property or triggering the `UpdateGraph` event.

Here is the `fcneval.isSuitable` method:

```
function isOk = isSuitable(funcH)
   v = [1 1;1 1];
   % Can the expression except 2 numeric inputs
   try
      funcH(v,v);
   catch
      me = MException('DocExample:fcneval',...
         ['The function ',func2str(funcH),' Is not a suitable F(x,y)']);
      isOk = me;
      return
   end
   % Does the expression return non-scalar data
   if isscalar(funcH(v,v));
      me = MException('DocExample:fcneval',...
         ['The function ',func2str(funcH),'' Returns a scalar when evaluated']);
      isOk = me;
      return
   end
   isOk = [];
end
```

The `fcneval.isSuitable` method could provide additional test to ensure that the expression assigned to the `FofXY` property meets the criteria required by the class design.

Other Approaches

The class could have implemented a property set event for the `FofXY` property and would, therefore, not need to call `notify` (see “Listen for Changes to Property Values” on page 11-32). Defining a class event provides more flexibility in this case because you can better control event triggering.

For example, suppose that you wanted to update the graph only if the new data is different. If the new expression produced the same data within some tolerance, the `set.FofXY` method could not trigger the event and avoid updating the graph. However, the method could still set the property to the new value.

Listener and Callback for UpdateGraph Event

The `fcnview` class creates a listener for the `UpdateGraph` event using the `addlistener` method:

```
obj.HLUpdateGraph = addlistener(obj.FcnObject,'UpdateGraph',...
          @(src,evnt)listenUpdateGraph(obj,src,evnt)); % Add obj to argument list
```

The `fcnview` object stores a handle to the `event.listener` object in its `HLUpdateGraph` property, which is used to enable/disable the listener by a context menu (see “Enable and Disable Listeners” on page 11-50).

The `fcnview` object (`obj`) is added to the two default arguments (`src`, `evnt`) passed to the listener callback. Keep in mind, the source of the event (`src`) is the `fcneval` object, but the `fcnview` object contains the handle of the surface object that the callback updates.

The `listenUpdateGraph` function is defined as follows:

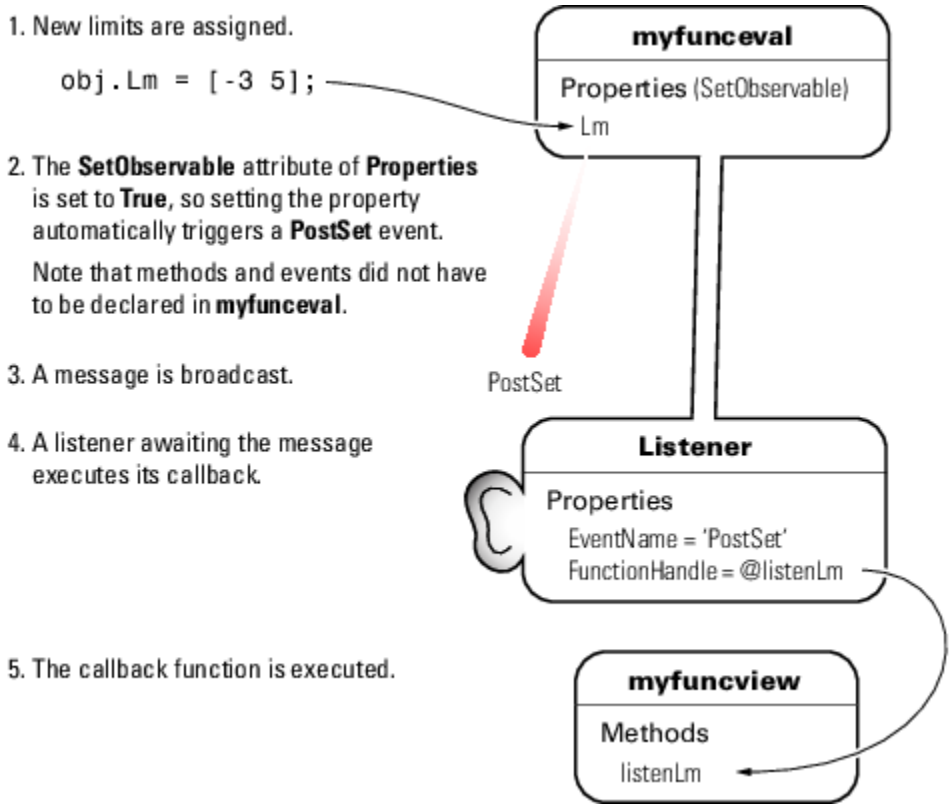
```
function listenUpdateGraph(obj,src,evnt)
   if ishandle(obj.HSurface) % If surface exists
      obj.updateSurfaceData % Update surface data
   end
end
```

The `updateSurfaceData` function is a class method that updates the surface data when a different mathematical function is assigned to the `fcneval` object. Updating a graphics object data is more efficient than creating a new object using the new data:

```
function updateSurfaceData(obj)
% Get data from fcneval object and set surface data
   set(obj.HSurface,...
      'XData',obj.FcnObject.Data.X,...
      'YData',obj.FcnObject.Data.Y,...
      'ZData',obj.FcnObject.Data.Matrix);
end
```

The PostSet Event Listener

All properties support the predefined `PostSet` event (See “Property-Set and Query Events” on page 11-14 for more information on property events). This example uses the `PostSet` event for the `fcneval` `Lm` property. This property contains a two-element vector specifying the range over which the mathematical function is evaluated. Just after this property is changed (by a statement like `obj.Lm = [-3 5];`), the `fcnview` objects listening for this event update the graph to reflect the new data.



Sequence During the Lm Property Assignment

The `fcneval` class defines a set function for the `Lm` property. When a value is assigned to this property during object construction or property reassignment, the following sequence occurs:

1. An attempt is made to assign argument value to `Lm` property.
2. The `set.Lm` method executes to check whether the value is in appropriate range — if yes, it makes assignment, if no, it generates an error.
3. If the value of `Lm` is set successfully, MATLAB triggers a `PostSet` event.
4. All listeners execute their callbacks, but the order is nondeterministic.

The `PostSet` event does not occur until an actual assignment of the property occurs. The property set function provides an opportunity to deal with potential assignment errors before the `PostSet` event occurs.

Enable PostSet Property Event

To create a listener for the `PostSet` event, you must set the property's `SetObservable` attribute to `true`:

```
properties (SetObservable = true)
   Lm = [-2*pi 2*pi];  % specifies default value
end
end
```

MATLAB automatically triggers the event so it is not necessary to call `notify`.

"Property Attributes" on page 8-8 provides a list of all property attributes.

Listener and Callback for PostSet Event

The `fcnview` class creates a listener for the `PostSet` event using the `addlistener` method:

```
obj.HLLm = addlistener(obj.FcnObject,'Lm','PostSet',...
          @(src,evnt)listenLm(obj,src,evnt)); % Add obj to argument list
```

The `fcnview` object stores a handle to the `event.listener` object in its `HLLm` property, which is used to enable/disable the listener by a context menu (see "Enable and Disable Listeners" on page 11-50).

The `fcnview` object (`obj`) is added to the two default arguments (`src`, `evnt`) passed to the listener callback. Keep in mind, the source of the event (`src`) is the `fcneval` object, but the `fcnview` object contains the handle of the surface object that the callback updates.

The callback sets the axes limits and updates the surface data because changing the limits causes the mathematical function to be evaluated over a different range:

```
function listenLm(obj,src,evnt)
   if ishandle(obj.HAxes) % If there is an axes
      lims(obj); % Update its limits
      if ishandle(obj.HSurface) % If there is a surface
         obj.updateSurfaceData % Update its data
      end
   end
end
```

Enable and Disable Listeners

Each `fcnview` object stores the handle of the listener objects it creates so that the listeners can be enabled or disabled via a context menu after the graphs are created. All listeners are instances of the `event.listener` class, which defines a property called `Enabled`. By default, this property has a value of `true`, which enables the listener. If you set this property to `false`, the listener still exists, but is disabled. This example creates a context menu active on the axes of each graph that provides a way to change the value of the `Enabled` property.

Context Menu Callback

There are two callbacks used by the context menu corresponding to the two items on the menu:

- **Listen** — Sets the `Enabled` property for both the `UpdateGraph` and `PostSet` listeners to `true` and adds a check mark next to the **Listen** menu item.
- **Don't Listen** — Sets the `Enabled` property for both the `UpdateGraph` and `PostSet` listeners to `false` and adds a check mark next to the **Don't Listen** menu item.

Both callbacks include the `fcnview` object as an argument (in addition to the required source and event data arguments) to provide access to the handle of the listener objects.

The `enableLisn` function is called when the user selects **Listen** from the context menu.

```
function enableLisn(obj,src,evnt)
   obj.HLUpdateGraph.Enabled = true; % Enable listener
   obj.HLLm.Enabled = true; % Enable listener
   set(obj.HEnableCm,'Checked','on') % Check Listen
```

```
   set(obj.HDisableCm,'Checked','off') % Uncheck Don't Listen
end
```

The `disableLisn` function is called when the user selects **Don't Listen** from the context menu.

```
function disableLisn(obj,src,evnt)
   obj.HLUpdateGraph.Enabled = false; % Disable listener
   obj.HLLm.Enabled = false; % Disable listener
   set(obj.HEnableCm,'Checked','off') % Uncheck Listen
   set(obj.HDisableCm,'Checked','on') % Check Don't Listen
end
```

@fcneval/fcneval.m Class Code

```
classdef fcneval < handle
   properties
      FofXY
   end

   properties (SetObservable = true)
      Lm = [-2*pi 2*pi]
   end % properties SetObservable = true

   properties (Dependent = true)
      Data
   end

   events
      UpdateGraph
   end

   methods
      function obj = fcneval(fcn_handle,limits)
         if nargin > 0
            obj.FofXY = fcn_handle;
            obj.Lm = limits;

         end
      end

      function set.FofXY(obj,func)
         me = fcneval.isSuitable(func);
         if ~isempty(me)
            throw(me)
         end
         obj.FofXY = func;
         notify(obj,'UpdateGraph');
      end

      function set.Lm(obj,lim)
         if  ~(lim(1) < lim(2))
            error('Limits must be monotonically increasing')
         else
            obj.Lm = lim;
         end
      end

      function data = get.Data(obj)
         [x,y] = fcneval.grid(obj.Lm);
         matrix = obj.FofXY(x,y);
         data.X = x;
         data.Y = y;
         data.Matrix = real(matrix);

      end

   end % methods
```

```
   methods (Static = true)
      function [x,y] = grid(lim)
         inc = (lim(2)-lim(1))/20;
         [x,y] = meshgrid(lim(1):inc:lim(2));
      end % grid

      function isOk = isSuitable(funcH)
         v = [1 1;1 1];
         try
            funcH(v,v);
         catch  %#ok<CTCH>
            me = MException('DocExample:fcneval',...
               ['The function ',func2str(funcH),' Is not a suitable F(x,y)']);
            isOk = me;
            return
         end
         if isscalar(funcH(v,v));
            me = MException('DocExample:fcneval',...
               ['The function ',func2str(funcH),' Returns a scalar when evaluated']);
            isOk = me;
            return
         end
         isOk = [];
      end

   end

end
```

@fcnview/fcnview.m Class Code

```
classdef fcnview < handle
   properties
      FcnObject     % fcneval object
      HAxes         % subplot axes handle
      HLUpdateGraph % UpdateGraph listener handle
      HLLm          % Lm property PostSet listener handle
      HEnableCm     % "Listen" context menu handle
      HDisableCm    % "Don't Listen" context menu handle
      HSurface      % Surface object handle
   end

   methods
      function obj = fcnview(fcnobj)
         if nargin > 0
            obj.FcnObject = fcnobj;
            obj.createLisn;
         end
      end

      function createLisn(obj)
         obj.HLUpdateGraph = addlistener(obj.FcnObject,'UpdateGraph',...
            @(src,evnt)listenUpdateGraph(obj,src,evnt));
         obj.HLLm = addlistener(obj.FcnObject,'Lm','PostSet',...
            @(src,evnt)listenLm(obj,src,evnt));
      end

      function lims(obj)
         lmts = obj.FcnObject.Lm;
         set(obj.HAxes,'XLim',lmts);
         set(obj.HAxes,'Ylim',lmts);
      end
```

```
      function updateSurfaceData(obj)
         data = obj.FcnObject.Data;
         set(obj.HSurface,...
            'XData',data.X,...
            'YData',data.Y,...
            'ZData',data.Matrix);
      end

      function listenUpdateGraph(obj,~,~)
         if ishandle(obj.HSurface)
            obj.updateSurfaceData
         end
      end

      function listenLm(obj,~,~)
         if ishandle(obj.HAxes)
            lims(obj);
            if ishandle(obj.HSurface)
               obj.updateSurfaceData
            end
         end
      end

      function delete(obj)
         if ishandle(obj.HAxes)
            delete(obj.HAxes);
         else
            return
         end
      end

   end
   methods (Static)
      createViews(a)
   end
end
```

@fcnview/createViews

```
function createViews(fcnevalobj)
   p = pi; deg = 180/p;
   hfig = figure('Visible','off',...
      'Toolbar','none');

   for k=4:-1:1
      fcnviewobj(k) = fcnview(fcnevalobj);
      axh = subplot(2,2,k);
      fcnviewobj(k).HAxes = axh;
      hcm(k) = uicontextmenu;
      set(axh,'Parent',hfig,...
         'FontSize',8,...
         'UIContextMenu',hcm(k))
      fcnviewobj(k).HEnableCm = uimenu(hcm(k),...
         'Label','Listen',...
         'Checked','on',...
         'Callback',@(src,evnt)enableLisn(fcnviewobj(k),src,evnt));
      fcnviewobj(k).HDisableCm = uimenu(hcm(k),...
         'Label','Don''t Listen',...
```

```
            'Checked','off',...
            'Callback',@(src,evnt)disableLisn(fcnviewobj(k),src,evnt));
         az = p/k*deg;
         view(axh,az,30)
         title(axh,['View: ',num2str(az),' 30'])
         fcnviewobj(k).lims;
         surfLight(fcnviewobj(k),axh)
      end
      set(hfig,'Visible','on')
   end
   function surfLight(obj,axh)
      obj.HSurface = surface(obj.FcnObject.Data.X,...
         obj.FcnObject.Data.Y,...
         obj.FcnObject.Data.Matrix,...
         'FaceColor',[.8 .8 0],'EdgeColor',[.3 .3 .2],...
         'FaceLighting','phong',...
         'FaceAlpha',.3,...
         'HitTest','off',...
         'Parent',axh);
      lims(obj)
      camlight left; material shiny; grid off
      colormap copper
   end

   function enableLisn(obj,~,~)
      obj.HLUpdateGraph.Enabled = true;
      obj.HLLm.Enabled = true;
      set(obj.HEnableCm,'Checked','on')
      set(obj.HDisableCm,'Checked','off')
   end

   function disableLisn(obj,~,~)
      obj.HLUpdateGraph.Enabled = false;
      obj.HLLm.Enabled = false;
      set(obj.HEnableCm,'Checked','off')
      set(obj.HDisableCm,'Checked','on')
   end
```

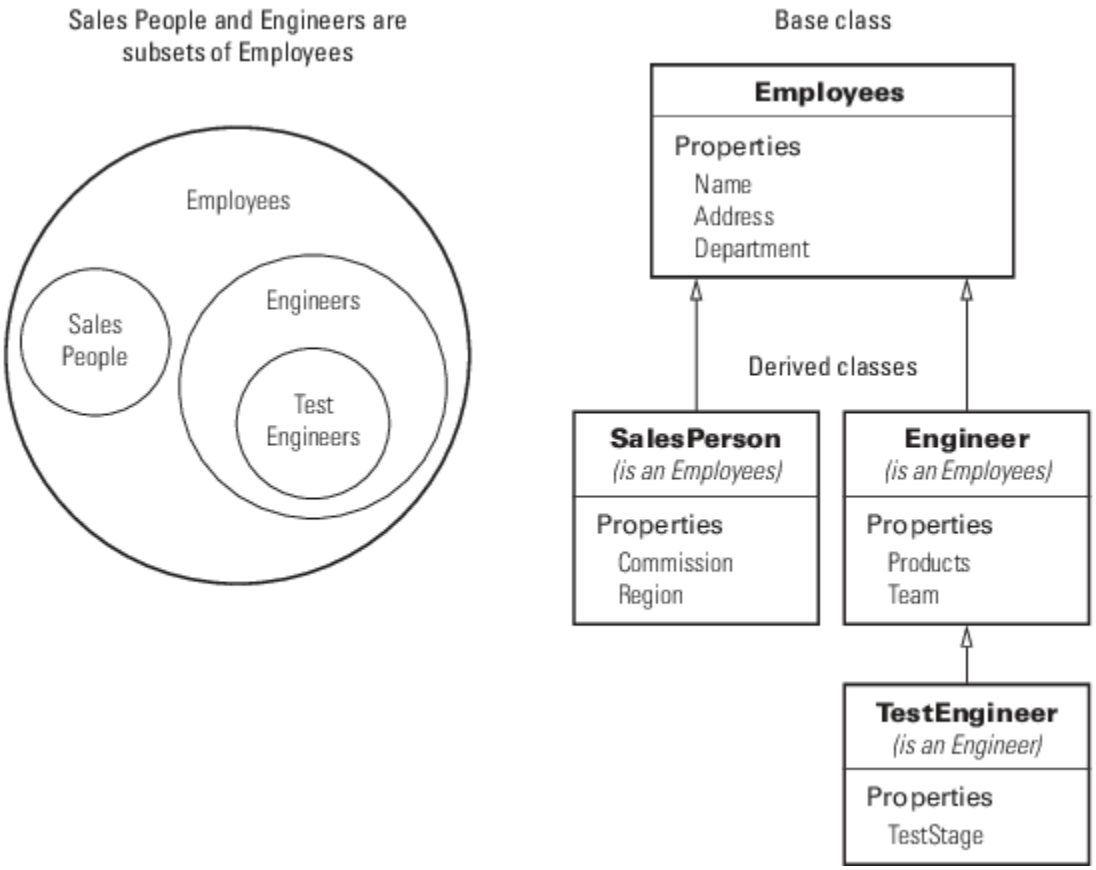
How to Build on Other Classes

Hierarchies of Classes — Concepts

In this section...

Classification

Organizing classes into hierarchies facilitates the reuse of code and the reuse of solutions to design problems that have already been solved. You can think of class hierarchies as sets — supersets (referred to as superclasses or base classes), and subsets (referred to as subclasses or derived classes). For example, the following picture shows how you could represent an employee database with classes.



The root of the hierarchy is the `Employees` class. It contains data and operations that apply to the set of all employees. Contained in the set of employees are subsets whose members, while still employees, are also members of sets that more specifically define the type of employee. Subclasses like `TestEngineer` are examples of these subsets.

Develop the Abstraction

Classes are representations of real world concepts or things. When designing a class, form an abstraction of what the class represents. Consider an abstraction of an employee and what are the essential aspects of employees for the intended use of the class. Name, address, and department can be what all employees have in common.

When designing classes, your abstraction contains only those elements that are necessary. For example, the employee hair color and shoe size certainly characterize the employee, but are probably not relevant to the design of this employee class. Their sales region is relevant only to some employee so this characteristic belongs in a subclass.

Design of Class Hierarchies

As you design a system of classes, put common data and functionality in a superclass, which you then use to derive subclasses. The subclasses inherit the data and functionality of the superclass and define only aspects that are unique to their particular purposes. This approach provides advantages:

- Avoid duplicating code that is common to all classes.
- Add or change subclasses at any time without modifying the superclass or affecting other subclasses.
- If the superclass changes (for example, all employees are assigned a number), then the subclass automatically get these changes.

Super and Subclass Behavior

Subclass objects behave like objects of the superclass because they are specializations of the superclass. This fact facilitates the development of related classes that behave similarly, but are implemented differently.

A Subclass Object “Is A” Superclass Object

You can usually describe the relationship between an object of a subclass and an object of its superclass with a statement like:

The subclass is a superclass. For example: An Engineer is an Employee.

This relationship implies that objects belonging to a subclass have the same properties, methods, and events as the superclass. Subclass objects also have any new features defined by the subclass. Test this relationship with the `isa` function.

Treat Subclass Objects like Superclass Objects

You can pass a subclass object to a superclass method, but you can access only those properties that the superclass defines. This behavior enables you to modify the subclasses without affecting the superclass.

Two points about super and subclass behavior to keep in mind are:

- Methods defined in the superclass can operate on subclass objects.
- Methods defined in the subclass cannot operate on superclass objects.

Therefore, you can treat an `Engineer` object like any other `Employees` object, but an `Employee` object cannot pass for an `Engineer` object.

Limitations to Object Substitution

MATLAB determines the class of an object based on its most specific class. Therefore, an `Engineer` object is of `class` `Engineer`, while it is also an `Employees` object, as using the `isa` function reveals.

Generally, MATLAB does not allow you to create arrays containing a mix of superclass and subclass objects because an array can be of only one class. If you attempt to concatenate objects of different classes, MATLAB looks for a converter method defined by the less dominant class

See “Concatenating Objects of Different Classes” on page 10-17 for more information.

See `matlab.mixin.Heterogeneous` for information on defining heterogeneous class hierarchies.

See “Object Converters” on page 17-5 for information on defining converter methods.

Implementation and Interface Inheritance

MATLAB classes support both the inheritance of implemented methods from a superclass and the inheritance of interfaces defined by abstract methods in the superclass.

Implementation inheritance enables code reuse by subclasses. For example, an `employee` class can have a `submitStatus` method that all `employee` subclasses can use. Subclasses can extend an inherited method to provide specialized functionality, while reusing the common aspects. See “Modify Inherited Methods” on page 12-13 for more information on this process.

Interface inheritance is useful in these cases:

- You want a group of classes to provide a common interface.
- Subclasses create specialized implementations of methods and properties.

Create an interface using an abstract class as the superclass. This class defines the methods and properties that you must implement in the subclasses, but does not provide an implementation.

The subclasses must provide their own implementation of the abstract members of the superclass. To create an interface, define methods and properties as abstract using their `Abstract` attribute. See “Abstract Classes and Class Members” on page 12-68 for more information and an example.

See Also

Related Examples

- “Design Subclass Constructors” on page 12-7

Subclass Syntax

In this section...

Subclass Definition Syntax

To define a class that is a subclass of another class, add the superclass to the `classdef` line after a `<` character:

```
classdef ClassName < SuperClass
```

When inheriting from multiple classes, use the & character to indicate the combination of the superclasses:

```
classdef ClassName < SuperClass1 & SuperClass2
```

See “Class Member Compatibility” on page 12-19 for more information on deriving from multiple superclasses.

Class Attributes

Subclasses do not inherit superclass attributes.

Subclass double

Suppose you want to define a class that derives from `double` and restricts values to be positive numbers. The `PositiveDouble` class:

- Supports a default constructor (no input arguments). See “No Input Argument Constructor Requirement” on page 9-18
- Restricts the inputs to positive values using `mustBePositive`
- Calls the superclass constructor with the input value to create the double numeric value

```
classdef PositiveDouble < double
   methods
      function obj = PositiveDouble(data)
         if nargin == 0
            data = 1;
         else
            mustBePositive(data)
         end
         obj = obj@double(data);
      end
   end
end
```

Create an object of the `PositiveDouble` class using a 1-by-5 array of numbers:

```
a = PositiveDouble(1:5);
```

You can perform operations on objects of this class like any double.

```
sum(a)

ans =

    15
```

Objects of the `PositiveDouble` class must be positive values.

```
a = PositiveDouble(0:5);

Error using mustBePositive (line 19)
Value must be positive.

Error in PositiveDouble (line 7)
            mustBePositive(data)
```

See Also

Related Examples

- “Design Subclass Constructors” on page 12-7
- “Subclasses of MATLAB Built-In Types” on page 12-40

Design Subclass Constructors

In this section...

Call Superclass Constructor Explicitly

Explicitly calling each superclass constructor from a subclass constructor enables you to:

- Pass arguments to superclass constructors
- Control the order in which MATLAB calls the superclass constructors

If you do not explicitly call the superclass constructors from the subclass constructor, MATLAB implicitly calls these constructors with no arguments. The superclass constructors must support the no argument syntax to support implicit calls, and the constructors are called in the order they appear at the top of the class block, from left to right. To change the order in which MATLAB calls the constructors or to call constructors with arguments, call the superclass constructors explicitly from the subclass constructor.

If you do not define a subclass constructor, you can call the default constructor with superclass arguments. For more information, see “Default Constructor” on page 9-17 and “Implicit Call to Inherited Constructor” on page 9-21.

Call Superclass Constructor from Subclass

To call the constructor for each superclass within the subclass constructor, use the following syntax:

```
obj@SuperClass1(args,...);

...

obj@SuperclassN(args,...);
```

Where *`obj`* is the output of the subclass constructor, *`SuperClass...`* is the name of a superclass, and *`args`* are any arguments required by the respective superclass constructor.

For example, the following segment of a class definition shows that a class called `Stocks` that is a subclass of a class called `Assets`.

```
classdef Stocks < Assets
   methods
      function s = Stocks(asset_args,...)
         if nargin == 0
            % Assign values to asset_args
         end
         % Call asset constructor
         s@Assets(asset_args);
         ...
```

```
      end
   end
end
```

“Subclass Constructors” on page 9-18 provides more information on creating subclass constructor methods.

Reference Superclasses Contained in Packages

If a superclass is contained in a package, include the package name. For example, the `Assests` class is in the `finance` package:

```
classdef Stocks < finance.Assets
   methods
      function s = Stocks(asset_args,...)
         if nargin == 0
            ...
         end
         % Call asset constructor
         s@finance.Assets(asset_args);
         ...
      end
   end
end
```

Initialize Objects Using Multiple Superclasses

To derive a class from multiple superclasses, initialize the subclass object with calls to each superclass constructor:

```
classdef Stocks < finance.Assets & Taxable
   methods
      function s = Stocks(asset_args,tax_args,...)
         if nargin == 0
            ...
         end
         % Call asset and member class constructors
         s@finance.Assets(asset_args)
         s@Taxable(tax_args)
         ...
      end
   end
end
```

Subclass Constructor Implementation

To ensure that your class constructor supports the zero arguments syntax, assign default values to input argument variables before calling the superclass constructor. You cannot conditionalize a subclass call to the superclass constructor. Locate calls to superclass constructors outside any conditional code blocks.

For example, the `Stocks` class constructor supports the no argument case with the `if` statement, but calls the superclass constructor outside of the `if` code block.

```
classdef Stocks < finance.Assets
   properties
      NumShares
```

```
      Symbol
   end
   methods
      function s = Stocks(description,numshares,symbol)
         if nargin == 0
            description = '';
            numshares = 0;
            symbol = '';
         end
         s@finance.Assets(description);
         s.NumShares = numshares;
         s.Symbol = symbol;
      end
   end
end
```

Call Only Direct Superclass from Constructor

Call only direct superclass constructors from a subclass constructor. For example, suppose class `B` derives from class `A` and class `C` derives from class `B`. The constructor for class `C` cannot call the constructor for class `A` to initialize properties. Class `B` must initialize class `A` properties.

The following implementations of classes `A`, `B`, and `C` show how to design this relationship in each class.

Class `A` defines properties `x` and `y`, but assigns a value only to `x`:

```
classdef A
   properties
      x
      y
   end
   methods
      function obj = A(x)
         ...
            obj.x = x;
      end
   end
end
```

Class `B` inherits properties `x` and `y` from class `A`. The class `B` constructor calls the class `A` constructor to initialize `x` and then assigns a value to `y`.

```
classdef B < A
   methods
      function obj = B(x,y)
         ...
         obj@A(x);
         obj.y = y;
      end
   end
end
```

Class `C` accepts values for the properties `x` and `y`, and passes these values to the class `B` constructor, which in turn calls the class `A` constructor:

```
classdef C < B
   methods
```

```
      function obj = C(x,y)
         ...
         obj@B(x,y);
      end
   end
end
```

See Also

Related Examples

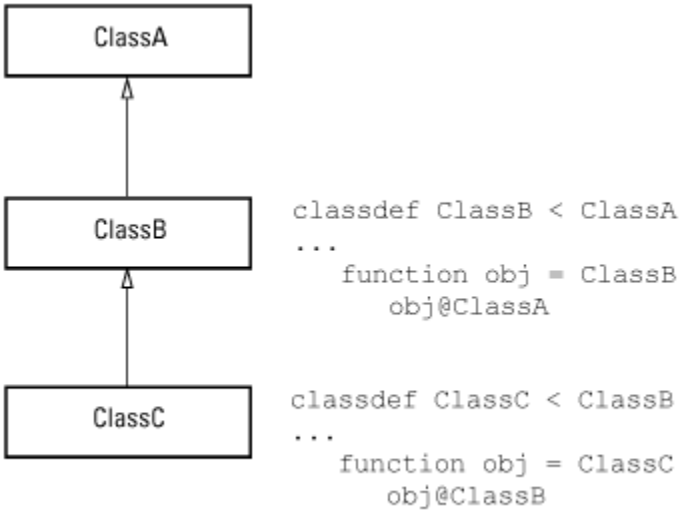
- “No Input Argument Constructor Requirement” on page 9-18

Control Sequence of Constructor Calls

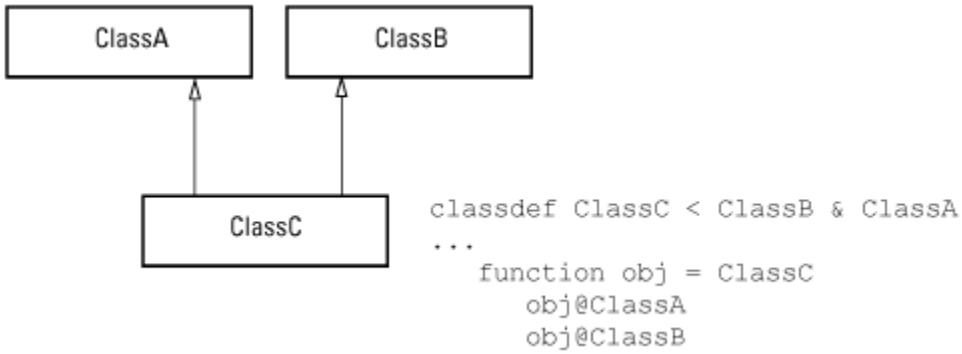
MATLAB does not guarantee the sequence in which superclass constructors are called when constructing a subclass object. However, you can control this order by calling superclass constructors explicitly from the subclass constructor.

If you explicitly call a superclass constructor from the most specific subclass constructor (`ClassC` in the following diagram), then MATLAB calls the most specific subclass constructor first. If you do not make an explicit call to a superclass constructor from the subclass constructor, MATLAB makes the implicit call when accessing the object.

Suppose that you have a hierarchy of classes in which `ClassC` derives from `ClassB`, which derives from `ClassA`. The constructor for a subclass can call only direct superclasses. Therefore, each class constructor can call the direct superclass constructor:



In cases of multiple inheritance, the subclass constructor can call each superclass constructor. To ensure that a specific superclass constructor calling sequence is followed, call all direct superclass constructors explicitly from the most specific subclass constructor:



See Also

Related Examples

- “Call Only Direct Superclass from Constructor” on page 12-9
- “Class Constructor Methods” on page 9-15

Modify Inherited Methods

In this section...

When to Modify Superclass Methods

Class designs enable you to pass subclass objects to superclass methods. The superclass method executes properly because the subclass object is a superclass object. However, subclasses can implement their own versions of superclass methods, which MATLAB calls when passed subclass objects.

Subclasses override inherited methods (that is, implement a method of the same name) when there is a need to provide specialized behavior in the subclass. Here are some patterns that override superclass methods.

- Extend the superclass method by calling it from within the subclass method. The subclass method can perform subclass-specific processing in addition to calling the superclass method.
- In a superclass method, implement a series of steps in a procedure using protected methods. Then reimplement these steps in a subclass method by redefining the protected methods that are called from within a public superclass method.
- Redefine the same-named method in the subclass, but use different implementations to perform the same operation differently on subclass objects.
- Implement abstract superclass methods in the subclass. Abstract superclasses can define methods with no implementation and rely on subclasses to provide the implementation. For more information, see “Define an Interface Superclass” on page 12-72.

Subclass methods that override superclass methods must define the same value for the `Access` attribute as is defined by the superclass method.

Extend Superclass Methods

Calling the same-named superclass method from a subclass method enables you to extend the superclass method for subclass objects without affecting the superclass method.

For example, suppose that both superclass and subclass define a method called `foo`. The subclass method calls the superclass method and performs other steps in addition to the call to the superclass method. The subclass method can operate on the specialized parts to the subclass that are not part of the superclass.

For example, this subclass defines a `foo` method that calls the superclass `foo` method

```
classdef Sub < Super
   methods
      function foo(obj)
```

```
         % preprocessing steps
          ...
         foo@Super(obj);
         % postprocessing steps
          ...
      end
   end
end
```

Reimplement Superclass Process in Subclass

A superclass method can define a process that executes in a series of steps using a method for each step (typically `Access` attribute is set to `protected` for the step methods). This pattern (referred to as a template method) enables subclasses to create their own versions of the protected methods that implement the individual steps in the process. The process is specialized for the subclass.

Implement this technique as shown here:

```
classdef Super
   methods (Sealed)
      function foo(obj)
         step1(obj) % Call step1
         step2(obj) % Call step2
         step3(obj) % Call step3
      end
   end
   methods (Access = protected)
      function step1(obj)
         % Superclass version
      end
      function step2(obj)
         % Superclass version
      end
      function step3(obj)
         % Superclass version
      end
   end
end
```

The subclass does not override the `foo` method. Instead it overrides only the protected methods that perform the series of steps (`step1(obj)`, `step2(obj)`, `step3(obj)`). This technique enables the subclass to specialize the actions taken by each step, but not control the order of the steps in the process. When you pass a subclass object to the superclass `foo` method, MATLAB calls the subclass step methods because of the dispatching rules. For more information on method dispatching, see “Method Invocation” on page 9-11.

```
classdef Sub < Super
   ...
   methods (Access = protected)
      function step1(obj)
         % Subclass version
      end
      function step2(obj)
         % Subclass version
      end
      function step3(obj)
```

```
         % Subclass version
      end
      ...
   end
end
```

Redefine Superclass Methods

You can completely redefine a superclass method in a subclass. In this case, both the superclass and the subclass would define a method with the same name. However, the implementation would be different and the subclass method would not call the superclass method. Creating independent versions of the same-named method can be necessary when the same operation requires different implementation on the superclass and subclass.

```
classdef Super
   methods
      function foo(obj)
         % Superclass implementation
      end
   end
end

classdef Sub < Super
   methods
      function foo(obj)
         % Subclass implementation
      end
   end
end
```

Implement Abstract Method in Subclass

Abstract methods have no implementation. Subclasses that inherit abstract methods must provide a subclass-specific implementation for the subclass to be a concrete class. For more information, see “Abstract Classes and Class Members” on page 12-68.

```
classdef Super
   methods (Abstract)
      foo(obj)
         % Abstract method without implementation
   end
end

classdef Sub < Super
   methods
      function foo(obj)
         % Subclass implementation of concrete method
      end
   end
end
```

See Also

Related Examples

- “Call Superclass Methods on Subclass Objects” on page 5-11
- “Abstract Classes and Class Members” on page 12-68
- “Modify Inherited Properties” on page 12-17

Modify Inherited Properties

In this section...

Superclass Property Modification

There are two separate conditions under which you can redefine superclass properties:

- The value of the superclass property `Abstract` attribute is `true`
- The values of the superclass property `SetAccess` and `GetAccess` attributes are `private`

If a superclass defines a property as abstract, the subclass must implement a concrete version of this property or the subclass is also abstract. Superclasses define abstract properties to create a consistent interface among subclasses.

If a superclass defines a property with private access, then only the superclass can access this property. The subclass can implement a different property with the same name.

Private Local Property Takes Precedence in Method

When superclass and subclass define a property with the same name, methods that refer to this property access the property of the class defining the method.

For example, if a subclass property has the same name as a superclass private property, and a method of the superclass references the property name, MATLAB accesses the property defined by the superclass.

Consider the following classes, `Super` and `Sub`:

```
classdef Super
   properties (Access = private)
      Prop = 2
   end
   methods
      function p = superMethod(obj)
         p = obj.Prop;
      end
   end
end

classdef Sub < Super
   properties
      Prop = 1
   end
end
```

If you create an instance of the `Sub` class and use it to call the superclass method, MATLAB accesses the private property of the superclass:

```
subObj = Sub
```

```
subObj =

  Sub with properties:

    Prop: 1
```

```
subObj.superMethod
```

```
ans =

     2
```

See Also

More About

- “Property Attributes” on page 8-8

Subclassing Multiple Classes

In this section...

Specify Multiple Superclasses

When inheriting from multiple classes, use the & character to indicate the combination of the superclasses:

```
classdef ClassName < SuperClass1 & SuperClass2
```

For more information on class syntax, see “Subclass Syntax” on page 12-5.

Class Member Compatibility

When you create a subclass derived from multiple superclasses, the subclass inherits the properties, methods, and events defined by all specified superclasses. If more than one superclass defines a property, method, or event having the same name, there must be an unambiguous resolution to the multiple definitions. You cannot derive a subclass from any two or more classes that define incompatible class members.

Here are various situations where you can resolve name and definition conflicts.

Property Conflicts

If two or more superclasses define a property with the same name, then at least one of the following must be true:

- All, or all but one of the properties must have their `SetAccess` and `GetAccess` attributes set to `private`
- The properties have the same definition in all superclasses (for example, when all superclasses inherited the property from a common base class)

Method Conflicts

If two or more superclasses define methods with the same name, then at least one of the following must be true:

- The method `Access` attribute is `private` so only the defining superclass can access the method.
- The method has the same definition in all subclasses. This situation can occur when all superclasses inherit the method from a common base class and none of the superclasses override the inherited definition.
- The subclass redefines the method to disambiguate the multiple definitions across all superclasses. Therefore, the superclass methods must not have their `Sealed` attribute set to `true`.
- Only one superclass defines the method as `Sealed`, in which case, the subclass adopts the sealed method definition.

- The superclasses define the methods as `Abstract` and rely on the subclass to define the method.

Event Conflicts

If two or more superclasses define events with the same name, then at least one of the following must be true:

- The event `ListenAccess` and `NotifyAccess` attributes must be `private`.
- The event has the same definition in all superclasses (for example, when all superclasses inherited the event from a common base class)

Multiple Inheritance

Resolving the potential conflicts involved when defining a subclass from multiple classes often reduces the value of this approach. For example, problems can arise when you enhance superclasses in future versions and introduce new conflicts.

Reduce potential problems by implementing only one unrestricted superclass. In all other superclasses, all methods are

- Abstract
- Defined by a subclass
- Inherited from the unrestricted superclass

When using multiple inheritance, ensure that all superclasses remain free of conflicts in definition.

See Also

Related Examples

- “Design Subclass Constructors” on page 12-7
- “Handle Compatible Classes” on page 12-31

Specify Allowed Subclasses

In this section...

Why Control Allowed Subclasses

A class definition can specify a list of classes that it allows as subclasses. Classes not in the list cannot be defined as subclass of the class. To specify the allowed subclasses, use the `AllowedSubclasses` class attribute.

The `AllowedSubclasses` attribute provides a design point between `Sealed` classes, which do not allow subclassing, and the default behavior, which does not restrict subclassing.

By controlling the allowed subclasses, you can create a sealed hierarchy of classes. That is, a system of classes that enables a specific set of classes to derive from specific base classes, but that does not allow unrestricted subclassing.

See “Define Sealed Hierarchy of Classes” on page 12-22 for more about this technique.

Specify Allowed Subclasses

Specify a list of one or more allowed subclasses in the `classdef` statement by assigning `meta.class` objects to the `AllowedSubclasses` attribute. Create the `meta.class` object referencing a specific class using the `?` operator and the class name:

```
classdef (AllowedSubclasses = ?ClassName) MySuperClass
   ...
end
```

Use a cell array of `meta.class` objects to define more than one allowed subclass:

```
classdef (AllowedSubclasses = {?ClassName1,?ClassName2,...?ClassNameN}) MySuperClass
   ...
end
```

Always use the fully qualified class name when referencing the class name:

```
classdef (AllowedSubclasses = ?Package.SubPackage.ClassName1) MySuperClass
   ...
end
```

Assigning an empty cell array to the `AllowedSubclasses` attribute is effectively the same as defining a `Sealed` class.

```
classdef (AllowedSubclasses = {}) MySuperClass
   ...
end
```

Note Use only the `?` operator and the class name to generate `meta.class` objects. Values assigned to the `AllowedSubclasses` attribute cannot contain any other MATLAB expressions, including functions that return either `meta.class` objects or cell arrays of `meta.class` objects.

Result of Declaring Allowed Subclasses

Including a class in the list of `AllowedSubclasses` does not define that class as a subclass or require you to define the class as a subclass. It just allows the referenced class to be defined as a subclass. Declaring a class as an allowed subclass also does not affect whether this class can itself be subclassed.

A class definition can contain assignments to the `AllowedSubclasses` attribute that reference classes that are not currently defined or available on the MATLAB path. Any referenced subclass that MATLAB cannot find when loading the class is effectively removed from the list without causing an error or warning. MATLAB remembers the referenced class in case it becomes available at a later point in time.

Note If MATLAB does not find any of the classes in the allowed subclasses list, the class is effectively `Sealed`. A sealed class is equivalent to `AllowedSubclasses = {}`.

Use the `meta.class` property `RestrictsSubclassing` to determine if a class is `Sealed` or specifies `AllowedSubclasses`.

Define Sealed Hierarchy of Classes

The `AllowedSubclasses` attribute enables you to define a sealed class hierarchy by sealing the allowed subclasses:

```
classdef (AllowedSubclasses = {?SubClass1,?SubClass2}) SuperClass
   ...
end
```

Define the allowed subclasses as `Sealed`:

```
classdef (Sealed) SubClass1
   ...
end

classdef (Sealed) SubClass2
   ...
end
```

Sealed class hierarchies enable you to use the level of abstraction that your design requires while maintaining a closed system of classes.

See Also

Related Examples

- “Handle Compatible Classes” on page 12-31

Class Members Access

In this section...

Basic Knowledge

The material presented in this section builds on an understanding of these concepts:

Terminology and Concepts

- Class members — Properties, methods, and events defined by a class
- Defining class — The class defining the class member for which access is being specified
- Get access — Permission to read the value of a property, controlled by the property `GetAccess` attribute
- Set access — Permission to assign a value to a property; controlled by the property `SetAccess` attribute
- Method access – Determines what other methods and functions can call the class method; controlled by the method `Access` attribute
- Listen access — Permission to define listeners; controlled by the event `ListenAccess` attribute
- Notify access — Permission to trigger events, controlled by the event `NotifyAccess` attribute

Possible Values for Access to Class Members

The following class member attributes can contain a list of classes:

- Properties — `Access`, `GetAccess`, and `SetAccess`. For a list of all property attributes, see “Property Attributes” on page 8-8 .
- Methods — `Access`. For a list of all method attributes, see “Method Attributes” on page 9-4 .
- Events — `ListenAccess` and `NotifyAccess`. For a list of all event attributes, see “Event Attributes” on page 11-16.

These attributes accept these values:

- `public` — Unrestricted access
- `protected` — Access by defining class and its subclasses
- `private` — Access by defining class only
- Access list — A list of one or more classes. Only the defining class and the classes in the list have access to the class members to which the attribute applies. If you specify a list of classes, MATLAB does not allow access by any other class (that is, access is `private`, except for the listed classes).

Applications for Access Control Lists

Access control lists enable you to control access to specific class properties, methods, and events. Access control lists specify a list of classes to which you grant access to these class members.

This technique provides greater flexibility and control in the design of a system of classes. For example, use access control lists to define separate classes, but not allow access to class members from outside the class system.

Specify Access to Class Members

Specify the classes that are allowed to access a particular class member in the member access attribute statement. For example:

```
methods (Access = {?ClassName1,?ClassName2,...})
```

Use the class `meta.class` object to refer to classes in the access list. To specify more than one class, use a cell array of `meta.class` objects. Use the package name when referring to classes that are in packages.

Note Specify the `meta.class` objects explicitly (created with the `?` operator), not as values returned by functions or other MATLAB expressions.

How MATLAB Interprets Attribute Values

- Granting access to a list of classes restricts access to only:
  - The defining class
  - The classes in the list
  - Subclasses of the classes in the list
- Including the defining class in the access list gives all subclasses of the defining class access.
- MATLAB resolves references to classes in the access list only when the class is loaded. If MATLAB cannot find a class that is included in the access list, that class is effectively removed from access.
- MATLAB replaces unresolved `meta.class` entries in the list with empty `meta.class` objects.
- An empty access list (that is, an empty cell array) is equivalent to `private` access.

Specify Metaclass Objects

Generate the `meta.class` objects using only the `?` operator and the class name. Values assigned to the attributes cannot contain any other MATLAB expressions, including functions that return allowed attribute values:

- `meta.class` objects
- Cell arrays of `meta.class` objects
- The values `public`, `protected`, or `private`

Specify these values explicitly, as shown in the example code in this section.

Properties with Access Lists

These sample classes show the behavior of a property that grants read access (`GetAccess`) to a class. The `GrantAccess` class gives `GetAccess` to the `NeedAccess` class for the `Prop1` property:

```
classdef GrantAccess
   properties (GetAccess = ?NeedAccess)
      Prop1 = 7
   end
end
```

The `NeedAccess` class defines a method that uses the value of the `GrantAccess` `Prop1` value. The `dispObj` method is defined as a `Static` method, however, it could be an ordinary method.

```
classdef NeedAccess
   methods (Static)
      function dispObj(GrantAccessObj)
         disp(['Prop1 is: ',num2str(GrantAccessObj.Prop1)])
      end
   end
end
```

Get access to `Prop1` is private so MATLAB returns an error if you attempt to access the property from outside the class definition. For example, from the command line:

```
a = GrantAccess;
a.Prop1
```

```
Getting the 'Prop1' property of the 'GrantAccess' class is not allowed.
```

However, MATLAB allows access to `Prop1` by the `NeedAccess` class:

```
NeedAccess.dispObj(a)
```

```
Prop1 is: 7
```

Methods with Access Lists

Classes granted access to a method can:

- Call the method using an instance of the defining class.
- Define their own method with the same name (if not a subclass).
- Override the method in a subclass only if the superclass defining the method includes itself or the subclass in the access list.

These sample classes show the behavior of methods called from methods of other classes that are in the access list. The class `AcListSuper` gives the `AcListNonSub` class access to its `m1` method:

```
classdef AcListSuper
   methods (Access = {?AcListNonSub})
      function obj = m1(obj)
         disp ('Method m1 called')
      end
   end
end
```

Because `AcListNonSub` is in the access list of `m1`, its methods can call `m1` using an instance of `AcListSuper`:

```
classdef AcListNonSub
   methods
      function obj = nonSub1(obj,AcListSuper_Obj)
         % Call m1 on AcListSuper class
         AcListSuper_Obj.m1;
      end
      function obj = m1(obj)
         % Define a method named m1
         disp(['Method m1 defined by ',class(obj)])
      end
   end
end
```

Create objects of both classes:

```
a = AcListSuper;
b = AcListNonSub;
```

Call the `AcListSuper` `m1` method using an `AcListNonSub` method:

```
b.nonSub1(a);
```

```
Method m1 called
```

Call the `AcListNonSub` `m1` method:

```
b.m1;
```

```
Method m1 defined by AcListNonSub
```

Subclasses Without Access

Including the defining class in the access list for a method grants access to all subclasses derived from that class. When you derive from a class that has a method with an access list and that list does *not* include the defining class:

- Subclass methods cannot call the superclass method.
- Subclass methods can call the superclass method indirectly using an instance of a class that is in the access list.
- Subclasses cannot override the superclass method.
- Methods of classes that are in the superclass method access list, but that are not subclasses, can call the superclass method.

For example, `AcListSub` is a subclass of `AcListSuper`. The `AcListSuper` class defines an access list for method `m1`. However, this list does not include `AcListSuper`, so subclasses of `AcListSuper` do not have access to method `m1`:

```
classdef AcListSub < AcListSuper
   methods
      function obj = sub1(obj,AcListSuper_Obj)
         % Access m1 via superclass object (***NOT ALLOWED***)
         AcListSuper_Obj.m1;
      end
      function obj = sub2(obj,AcListNonSub_Obj,AcListSuper_obj)
         % Access m1 via object that is in access list (is allowed)
         AcListNonSub_Obj.nonSub1(AcListSuper_Obj);
```

```
      end
   end
end
```

No Direct Call to Superclass Method

Attempting to call the superclass `m1` method from the `sub1` method results in an error because subclasses are not in the access list for `m1`:

```
a = AcListSuper;
c = AcListSub;
c.sub1(a);
```

```
Cannot access method 'm1' in class 'AcListSuper'.

Error in AcListSub/sub1 (line 4)
         AcListSuper_Obj.m1;
```

Indirect Call to Superclass Method

You can call a superclass method from a subclass that does not have access to that method using an object of a class that is in the superclass method access list.

The `AcListSub` `sub2` method calls a method of a class (`AcListNonSub`) that is on the access list for `m1`. This method, `nonSub1`, does have access to the superclass `m1` method:

```
a = AcListSuper;
b = AcListNonSub;
c = AcListSub;
c.sub2(b,a);
```

```
Method m1 called
```

No Redefining Superclass Method

When subclasses are not included in the access list for a method, those subclasses cannot define a method with the same name. This behavior is not the same as cases in which the method `Access` is explicitly declared as `private`.

For example, adding the following method to the `AcListSub` class definition produces an error when you attempt to instantiate the class.

```
methods (Access = {?AcListNonSub})
   function obj = m1(obj)
      disp('AcListSub m1 method')
   end
end
```

```
c = AcListSub;
```

```
Class 'AcListSub' is not allowed to override the method 'm1' because neither it nor its
superclasses have been granted access to the method by class 'AcListSuper'.
```

Call Superclass from Listed Class Via Subclass

The `AcListNonSub` class is in the `m1` method access list. This class can define a method that calls the `m1` method using an object of the `AcListSub` class. While `AcListSub` is not in the access list for method `m1`, it is a subclass of `AcListSuper`.

For example, add the following method to the `AcListNonSub` class:

```
methods
   function obj = nonSub2(obj,AcListSub_Obj)
      disp('Call m1 via subclass object:')
      AcListSub_Obj.m1;
   end
end
```

Calling the `nonSub2` method results in execution of the superclass `m1` method:

```
b = AcListNonSub;
c = AcListSub;
b.nonSub2(c);
```

```
Call m1 via subclass object:
Method m1 called
```

This behavior is consistent with the behavior of any subclass object, which can substitute for an object of its superclass.

Abstract Methods with Access Lists

A class containing a method declared as `Abstract` is an abstract class. It is the responsibility of subclasses to implement the abstract method using the function signature declared in the class definition.

When an abstract method has an access list, only the classes in the access list can implement the method. A subclass that is not in the access list cannot implement the abstract method so that subclass is itself abstract.

See Also

Related Examples

- “Property Attributes” on page 8-8
- “Method Access List” on page 12-29
- “Event Access List” on page 12-30

Method Access List

This class declares an access list for the method `Access` attribute:

```
classdef MethodAccess
   methods (Access = {?ClassA, ?ClassB, ?MethodAccess})
      function listMethod(obj)
      ...
      end
   end
end
```

The `MethodAccess` class specifies the following method access:

- Access to `listMethod` from an instance of `MethodAccess` by methods of the classes `ClassA` and `ClassB`.
- Access to `listMethod` from an instance of `MethodAccess` by methods of subclasses of `MethodAccess`, because of the inclusion of `MethodAccess` in the access list.
- Subclasses of `ClassA` and `ClassB` are allowed to define a method named `listMethod`, and `MethodAccess` is allowed to redefine `listMethod`. However, if `MethodAccess` was not in the access list, its subclasses could not redefine `listMethod`.

See Also

Related Examples

- “Methods with Access Lists” on page 12-25

Event Access List

This class declares an access list for the event `ListenAccess` attribute:

```
classdef EventAccess
   events (NotifyAccess = private, ListenAccess = {?ClassA, ?ClassB})
      Event1
      Event2
   end
end
```

The class `EventAccess` specifies the following event access:

- Limits notify access for `Event1` and `Event2` to `EventAccess`; only methods of the `EventAccess` can trigger these events.
- Gives listen access for `Event1` and `Event2` to methods of `ClassA` and `ClassB`. Methods of `EventAccess`, `ClassA`, and `ClassB` can define listeners for these events. Subclasses of `MyClass` cannot define listeners for these events.

See Also

Related Examples

- “Events and Listeners Syntax” on page 11-18

Handle Compatible Classes

In this section...

Basic Knowledge

The material presented in this section builds on knowledge of the following information.

- “Design Subclass Constructors” on page 12-7
- “Subclassing Multiple Classes” on page 12-19
- “Comparison of Handle and Value Classes” on page 7-2

Key Concepts

Handle-compatible class — a class that you can include with handle classes in a class hierarchy, even if the class is not a handle class.

- All handle classes are handle-compatible.
- All superclasses of handle-compatible classes must also be handle compatible.

`HandleCompatible` — the class attribute that defines nonhandle classes as handle compatible.

When to Use Handle-Compatible Classes

Typically, when deriving a MATLAB class from other classes, all the superclasses are handle classes, or none of them are handle classes. However, there are situations in which a class provides some utility that is used by both handle and nonhandle subclasses. Because it is not legal to combine handle and nonhandle classes, the author of the utility class must implement two distinct versions of the utility.

The solution is to use handle-compatible classes. You can use handle-compatible classes with handle classes when forming sets of superclasses. Designate a class as handle compatible by using the `HandleCompatible` class attribute.

```
classdef (HandleCompatible) MyClass
   ...
end
```

Handle Compatibility Rules

Handle-compatible classes (that is, classes whose `HandleCompatible` attribute is set to `true`) follow these rules:

- All superclasses of a handle-compatible class must also be handle compatible
- If a class explicitly sets its `HandleCompatibility` attribute to `false`, then none of the class superclasses can be handle classes.

- If a class does not explicitly set its `HandleCompatible` attribute and, if any superclass is a handle, then all superclasses must be handle compatible.
- The `HandleCompatible` attribute is not inherited.

A class that does not explicitly set its `HandleCompatible` attribute to `true` is:

- A handle class if any of its superclasses are handle classes
- A value class if none of the superclasses are handle classes

Identify Handle Objects

To determine if an object is a handle object, use the `isa` function:

```
isa(obj,'handle')
```

See Also

Related Examples

- “How to Define Handle-Compatible Classes” on page 12-33

How to Define Handle-Compatible Classes

In this section...

What Is Handle Compatibility?

A class is handle compatible if:

- It is a handle class
- Its `HandleCompatible` attribute is set to `true`

The `HandleCompatible` class attribute identifies classes that you can combine with handle classes when specifying a set of superclasses.

Handle compatibility provides greater flexibility when defining abstract superclasses. For example, when using superclasses that support both handle and value subclasses, handle compatibility removes the need to define both a handle version and a nonhandle version of a class.

A Handle Compatible Class

In this example, the `Utility` class defines a method to reset property values to the default values defined in the respective class definition. This functionality is useful to both handle and value subclasses.

```
classdef (HandleCompatible) Utility
   methods
      function obj = resetDefaults(obj)
         mc = metaclass(obj);
         mp = mc.PropertyList;
         for k=1:length(mp)
            if mp(k).HasDefault && ~strcmp(mp(k).SetAccess,'private')
               obj.(mp(k).Name) = mp(k).DefaultValue;
            end
         end
      end
   end
end
```

The `Utility` class is handle compatible. Therefore, you can use it in the derivation of classes that are either handle classes or value classes. See “Class Introspection and Metadata” for information on using meta-data classes.

Return Modified Objects

The `resetDefaults` method defined by the `Utility` class returns the object it modifies. When you call `resetDefaults` with a value object, the method must return the modified object. It is important to implement methods that work with both handle and value objects in a handle compatible superclass. See “Object Modification” on page 5-35 for more information on modifying handle and value objects.

Consider the behavior of a value class that subclasses the `Utility` class. The `PropertyDefaults` class defines three properties, all of which have default values:

```
classdef PropertyDefaults < Utility
   properties
      p1 = datestr(rem(now,1)) % Current time
      p2 = 'red'               % Character vector
      p3 = pi/2                % Result of division operation
   end
end
```

Create a `PropertyDefaults` object. MATLAB evaluates the expressions assigned as default property values when the class is first loaded. MATLAB uses these same default values whenever you create an instance of this class in the current MATLAB session.

```
pd = PropertyDefaults
```

```
pd =

  PropertyDefaults with properties:

    p1: ' 4:42 PM'
    p2: 'red'
    p3: 1.5708
```

Assign new values that are different from the default values:

```
pd.p1 = datestr(rem(now,1));
pd.p2 = 'green';
pd.p3 = pi/4;
```

All `pd` object property values now contain values that are different from the default values originally defined by the class:

```
pd
```

```
pd =

  PropertyDefaults with properties:

    p1: ' 4:45 PM'
    p2: 'green'
    p3: 0.7854
```

Call the `resetDefaults` method, which is inherited from the `Utility` class. Because the `PropertyDefaults` class is not a handle class, return the modified object.

```
pd = pd.resetDefaults
```

```
pd =

  PropertyDefaults with properties:

    p1: ' 4:54 PM'
    p2: 'red'
    p3: 1.5708
```

If the `PropertyDefaults` class was a handle class, then you would not need to save the object returned by the `resetDefaults` method. To design a handle compatible class like `Utility`, ensure that all methods work with both kinds of classes.

Subclassing Handle-Compatible Classes

According to the rules described in “Handle Compatibility Rules” on page 12-31, when you combine a handle superclass with a handle-compatible superclass, the result is a handle subclass, which is handle compatible.

However, subclassing a handle-compatible class does not necessarily result in the subclass being handle compatible. Consider the following two cases, which demonstrate two possible results.

Combine Nonhandle Utility Class with Handle Classes

Define a class that subclasses a handle class and the handle-compatible `Utility` class discussed in “A Handle Compatible Class” on page 12-33. The `HPropertyDefaults` class has these characteristics:

- It is a handle class because it derives from `handle`.
- All its superclasses are handle compatible because handle classes are handle compatible by definition.

```
classdef HPropertyDefaults < handle & Utility
   properties
      GraphPrim = line
      Width = 1.5
      Color = 'black'
   end
end
```

The `HPropertyDefaults` class is handle compatible.

```
hpd = HPropertyDefaults;
mc = metaclass(hpd);
mc.HandleCompatible
```

```
ans =

     1
```

Nonhandle Subclasses of a Handle-Compatible Class

If you subclass both a value class that is not handle compatible and a handle compatible class, the subclass is a nonhandle compatible value class. The `ValueSub` class:

- Is a value class (it does not derive from `handle`.)
- One of its superclasses is handle compatible (the `Utility` class).

```
classdef ValueSub < MException & Utility
   methods
      function obj = ValueSub(str1,str2)
         obj = obj@MException(str1,str2);
      end
   end
end
```

The `ValueSub` class is a nonhandle-compatible value class because the `MException` class does not define the `HandleCompatible` attribute as `true`:

```
hv = ValueSub('MATLAB:narginchk:notEnoughInputs',...
      'Not enough input arguments.');
mc = metaclass(hv);
mc.HandleCompatible

ans =

     0
```

See Also

Related Examples

- “Methods for Handle-Compatible Classes” on page 12-37

Methods for Handle-Compatible Classes

In this section...

Methods for Handle and Value Objects

Objects passed to methods of handle-compatible classes can be either handle or value objects. There are two different behaviors to consider when implementing methods for a class that operate on both handles and values:

- If an input object is a handle object and the method alters the handle object, these changes are visible to all workspaces that contain the same handle.
- If an input object is a value object, then changes to the object made inside the method affect only the value inside the method workspace.

Handle compatible methods generally do not alter input objects because the effects of such changes are not the same for handle and nonhandle objects.

See “Object Modification” on page 5-35 for information about modifying handle and value objects.

Modify Value Objects in Methods

If a method operates on both handle and value objects, the method must return the modified object. For example, the `setTime` method returns the object it modifies:

```
classdef (HandleCompatible) Util
   % Utility class that adds a time stamp
   properties
      TimeStamp
   end
   methods
      function obj = setTime(obj)
         obj.TimeStamp = now;
      end
   end
end
```

See Also

Related Examples

- “Handle-Compatible Classes and Heterogeneous Arrays” on page 12-38

Handle-Compatible Classes and Heterogeneous Arrays

In this section...

Heterogeneous Arrays

A heterogeneous array contains objects of different classes. Members of a heterogeneous array have a common superclass, but can belong to different subclasses. See the `matlab.mixin.Heterogeneous` class for more information on heterogeneous arrays. The `matlab.mixin.Heterogeneous` class is a handle-compatible class.

Methods Must Be Sealed

You can invoke only those methods that are sealed by the common superclass on heterogeneous arrays (`Sealed` attribute set to `true`). Sealed methods prevent subclasses from overriding those methods and guarantee that methods called on heterogeneous arrays have the same definition for the entire array.

Subclasses cannot override sealed methods. In situations requiring subclasses to specialize methods defined by a utility class, you can employ the design pattern referred to as the template method.

Template Technique

Suppose that you implement a handle-compatible class that works with heterogeneous arrays. This approach enables you to seal public methods, while providing a way for each subclass to specialize how the method works on each subclass instance. In the handle-compatible class:

- Define a sealed method that accepts a heterogeneous array as input.
- Define a protected, abstract method that each subclass must implement.
- Within the sealed method, call the overridden method for each array element.

Each subclass in the heterogeneous hierarchy implements a concrete version of the abstract method. The concrete method provides specialized behavior required by the particular subclass.

The `Printable` class shows how to implement a template method approach:

```
classdef (HandleCompatible) Printable
   methods(Sealed)
      function print(aryIn)
         n = numel(aryIn);
         for k=1:n
            printElement(aryIn(k));
         end
      end
   end
   methods(Access=protected, Abstract)
```

```
      printElement(objIn)
   end
end
```

See Also

Related Examples

- “Handle Compatible Classes” on page 12-31

Subclasses of MATLAB Built-In Types

In this section...

MATLAB Built-In Types

Built-in types represent fundamental kinds of data such as numeric arrays, logical arrays, and character arrays. Other built-in types like cell arrays and structures contain data belonging to any class.

Built-in types define methods that perform operations on objects of these classes. For example, you can perform operations on numeric arrays such as, sorting, arithmetic, and logical operations.

See “Fundamental MATLAB Classes” for more information on MATLAB built-in classes.

Note Defining a class with the same name as a built-in class is not supported.

Built-In Types You Can Subclass

You can subclass MATLAB numeric classes and the `logical` class. For a list of numeric types, see “Numeric Types”.

You cannot subclass any class that has its `Sealed` attribute set to `true`. To determine if the class is `Sealed`, query the class metadata:

```
mc = ?ClassName;
mc.Sealed
```

A value of `0` indicates that the class is not `Sealed` and can be subclasses.

Why Subclass Built-In Types

Subclass a built-in class to extend the operations that you can perform on a particular class of data. For example, when you want to:

- Perform unique operations on class data
- Use methods of the built-in class and other built-in functions directly with objects of the subclass. For example, you do not need to reimplement all the mathematical operators if you derived from a class such as `double` that defines these operators.

Which Functions Work with Subclasses of Built-In Types

Consider a class that defines enumerations. It can derive from an integer class and inherit methods that enable you to compare and sort values. For example, integer classes like `int32` support all the relational methods (`eq`, `ge`, `gt`, `le`, `lt`, `ne`).

To see a list of functions that the subclass has inherited as methods, use the `methods` function:

```
methods('SubclassName')
```

Generally, you can use an object of the subclass with any:

- Inherited methods
- Functions that normally accept input arguments of the same class as the superclass.

Behavior of Built-In Functions with Subclass Objects

When you define a subclass of a built-in class, the subclass inherits all the methods defined by that built-in class. MATLAB also provides additional methods to subclasses of built-in classes that override several built-in functions.

Built-in functions and methods that work on built-in classes can behave differently when called with subclasses of built-in classes. Their behavior depends on which function you are using and whether your subclass defines properties.

Behavior Categories

When you call an inherited method on a subclass of a built-in class, the result depends on the nature of the operation performed by the method. The behaviors of these methods fit into several categories.

- Operations on data values return objects of the superclass. For example, if you subclass `double` and perform addition on two subclass objects, MATLAB adds the numeric values and returns a value of class `double`.
- Operations on the orientation or structure of the data return objects of the subclass. Methods that perform these kinds of operations include, `reshape`, `permute`, `transpose`, and so on.
- Converting a subclass object to a built-in class returns an object of the specified class. Functions such as `uint32`, `double`, `char` work with subclass objects the same as they work with built-in objects.
- Comparing objects or testing for inclusion in a specific set returns logical or built-in objects, depending on the function. Functions such as `isequal`, `ischar`, `isobject` work with subclass objects the same as they work with superclass objects.
- Indexing expressions return objects of the subclass. If the subclass defines properties, then default indexing no longer works. The subclass must define its own indexing methods.
- Concatenation returns an object of the subclass. If the subclass defines properties, then default concatenation no longer works and the subclass must define its own concatenation methods.

To list the built-in functions that work with a subclass of a built-in class, use the `methods` function.

Built-In Subclasses That Define Properties

When a subclass of a built-in class defines properties, MATLAB no longer supports indexing and concatenation operations. MATLAB cannot use the built-in functions normally called for these operations because subclass properties can contain any data.

The subclass must define what indexing and concatenation mean for a class with properties. If your subclass needs indexing and concatenation functionality, then the subclass must implement the appropriate methods.

Methods for Indexing

To support indexing operations, the subclass must implement these methods:

- `subsasgn` — Implement dot notation and indexed assignments
- `subsref` — Implement dot notation and indexed references
- `subsindex` — Implement object as index value

Note Modular indexing mixin classes were introduced in R2021b, but these mixins are not compatible with subclasses of built-in classes. See “Code Patterns for subsref and subsasgn Methods” on page 17-9 for more information on how to implement `subsref`, `subsasgn`, and `subsindex`.

Methods for Concatenation

To support concatenation, the subclass must implement the following methods:

- `horzcat` — Implement horizontal concatenation of objects
- `vertcat` — Implement vertical concatenation of objects
- `cat` — Implement concatenation of object arrays along specified dimension

See Also

Related Examples

- “Subclasses of Built-In Types with Properties” on page 12-53
- “Subclasses of Built-In Types Without Properties” on page 12-47

Behavior of Inherited Built-In Methods

In this section...

Subclass double

Most built-in functions used with built-in classes are actually methods of the built-in class. For example, the `double` and `single` classes define several methods to perform arithmetic operations, indexing, matrix operation, and so on. All these built-in class methods work with subclasses of the built-in class.

Subclassing `double` enables your class to use features without implementing the methods that a MATLAB built-in class defines.

The `DocSimpleDouble` class subclasses the built-in `double` class.

```
classdef DocSimpleDouble < double
   methods
      function obj = DocSimpleDouble(data)
         if nargin == 0
            data = 0;
         end
         obj = obj@double(data);
      end
   end
end
```

Create an instance of the class `DocSimpleDouble`.

```
sc = DocSimpleDouble(1:10)
```

```
sc =
   1x10 DocSimpleDouble:
  double data:
     1     2     3     4     5     6     7     8     9    10
```

Call a method inherited from class `double` that operates on the data, such as `sum`. `sum` returns a `double` and, therefore, uses the `display` method of class `double`:

```
sum(sc)
```

```
ans =
    55
```

Index `sc` like an array of doubles. The returned value is the class of the subclass:

```
a = sc(2:4)
```

```
a =
  1x3 DocSimpleDouble:
```

```
  double data:
     2     3     4
```

Indexed assignment works the same as the built-in class. The returned value is the class of the subclass:

```
sc(1:5) = 5:-1:1

sc =
   1x10 DocSimpleDouble:
  double data:
     5     4     3     2     1     6     7     8     9    10
```

Calling a method that modifies the order of the data elements operates on the data, but returns an object of the subclass:

```
sc = DocSimpleDouble(1:10);
sc(1:5) = 5:-1:1;
a = sort(sc)

a =
   1x10 DocSimpleDouble:
  double data:
     1     2     3     4     5     6     7     8     9    10
```

Built-In Data Value Methods

When you call a built-in data value method on a subclass object, MATLAB uses the superclass part of the subclass object as inputs to the method. The value returned is same class as the built-in class. For example:

```
sc = DocSimpleDouble(1:10);
a = sin(sc);
class(a)

ans =

double
```

Built-In Data Organization Methods

This group of built-in methods reorders or reshapes the input argument array. These methods operate on the superclass part of the subclass object, but return an object of the same type as the subclass.

```
sc = DocSimpleDouble(randi(9,1,10))

sc = DocSimpleDouble(randi(9,1,10))

sc =

  1x10 DocSimpleDouble:

  double data:
     6     1     8     9     7     7     7     4     6     2

b = sort(sc)
```

```
b =

  1x10 DocSimpleDouble:

  double data:
     1     2     4     6     6     7     7     7     8     9
```

Methods in this group include:

- `reshape`
- `permute`
- `sort`
- `transpose`
- `ctranspose`

Built-In Indexing Methods

When you use indexing to reference into an object that inherits from a built-in class, only the built-in data is referenced, not the properties defined by your subclass. You cannot make indexed references if your subclass defines properties, unless your subclass overrides the default `subsref` method.

For example, indexing element 2 in the `DocSimpleDouble` subclass object returns the second element in the vector.

```
sc = DocSimpleDouble(1:10);
a = sc(2)
```

```
a =
  DocSimpleDouble
  double data:
     2
```

The value returned from an indexing operation is an object of the subclass.

Assigning a new value to the second element in the `DocSimpleDouble` object operates only on the superclass data.

```
sc(2) = 12
```

```
sc =
   1x10 DocSimpleDouble:
  double data:
     1    12     3     4     5     6     7     8     9    10
```

The `subsref` method also implements dot notation for methods.

Built-In Concatenation Methods

Built-in classes use the functions `horzcat`, `vertcat`, and `cat` to implement concatenation. When you use these functions with subclass objects of the same type, MATLAB concatenates the superclass data to form a new object. For example, you can concatenate objects of the `DocSimpleDouble` class:

```
sc1 = DocSimpleDouble(1:10);
sc2 = DocSimpleDouble(11:20);
[sc1,sc2]
```

```
ans =
  1x20 DocSimpleDouble:
  double data:
  Columns 1 through 13
     1     2     3     4     5     6     7     8     9    10    11    12    13
  Columns 14 through 20
    14    15    16    17    18    19    20
```

```
[sc1;sc2]
```

```
ans =
  2x10 DocSimpleDouble:
  double data:
     1     2     3     4     5     6     7     8     9    10
    11    12    13    14    15    16    17    18    19    20
```

Concatenate two objects along a third dimension:

```
c = cat(3,sc1,sc2)
```

```
c =

  1x10x2 DocSimpleDouble:

  double data:

(:,:,1) =

     1     2     3     4     5     6     7     8     9    10


(:,:,2) =

    11    12    13    14    15    16    17    18    19    20
```

If the subclass of a built-in class defines properties, you cannot concatenate objects of the subclass. There is no way to determine how to combine properties of different objects. However, your subclass can define custom `horzcat` and `vertcat` methods to support concatenation in whatever way makes sense for your subclass.

See Also

Related Examples

- “Subclasses of Built-In Types Without Properties” on page 12-47
- “Subclasses of Built-In Types with Properties” on page 12-53

Subclasses of Built-In Types Without Properties

In this section...

Specialized Numeric Types

Subclass built-in numeric types to create customized data types that inherit the functionality of the built-in type. Add functionality to that provided by the superclass by implementing class methods. Subclasses without properties store numeric data as the superclass type. If your subclass design does not require properties to store other data, the implementation is simpler because you do not need to define indexing and concatenation methods.

For more information, see “Subclasses of MATLAB Built-In Types” on page 12-40.

A Class to Manage uint8 Data

This example shows a class derived from the built-in `uint8` class. This class simplifies the process of maintaining a collection of intensity image data defined by `uint8` values. The basic operations of the class include:

- Capability to convert various classes of image data to `uint8` to reduce object data storage.
- A method to display the intensity images contained in the subclass objects.
- Ability to use all the methods supported by `uint8` data (for example, `size`, indexing, `reshape`, `bitshift`, `cat`, `fft`, arithmetic operators, and so on).

The class data are matrices of intensity image data stored in the superclass part of the subclass object. This approach requires no properties.

The `DocUint8` class stores the image data, which converts the data, if necessary:

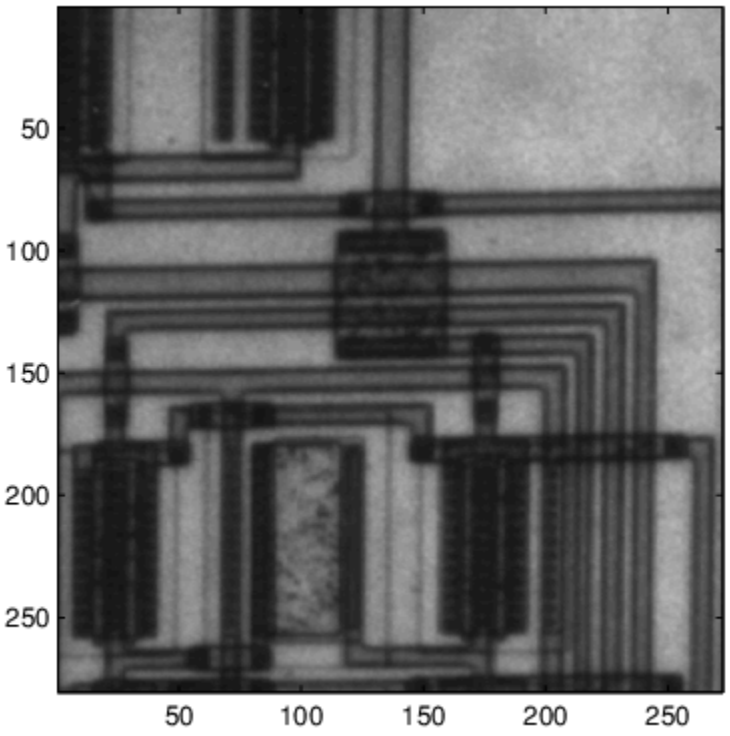
```
classdef DocUint8 < uint8
   methods
      function obj = DocUint8(data)
         if nargin == 0
            data = uint8(0);
         end
         obj = obj@uint8(data); % Store data on superclass
      end
      function h = showImage(obj)
         data = uint8(obj);
         figure; colormap(gray(256))
         h = imagesc(data,[0 255]);
         axis image
         brighten(.2)
      end
   end
end
```

Using the DocUint8 Class

Create DocUint8 Objects

The `DocUint8` class provides a method to display all images stored as `DocUint8` objects in a consistent way. For example:

```
cir = imread('circuit.tif');
img1 = DocUint8(cir);
img1.showImage;
```



Because `DocUint8` subclasses `uint8`, you can use any `uint8` methods. For example,

```
size(img1)
```

```
ans =
   280   272
```

returns the size of the image data.

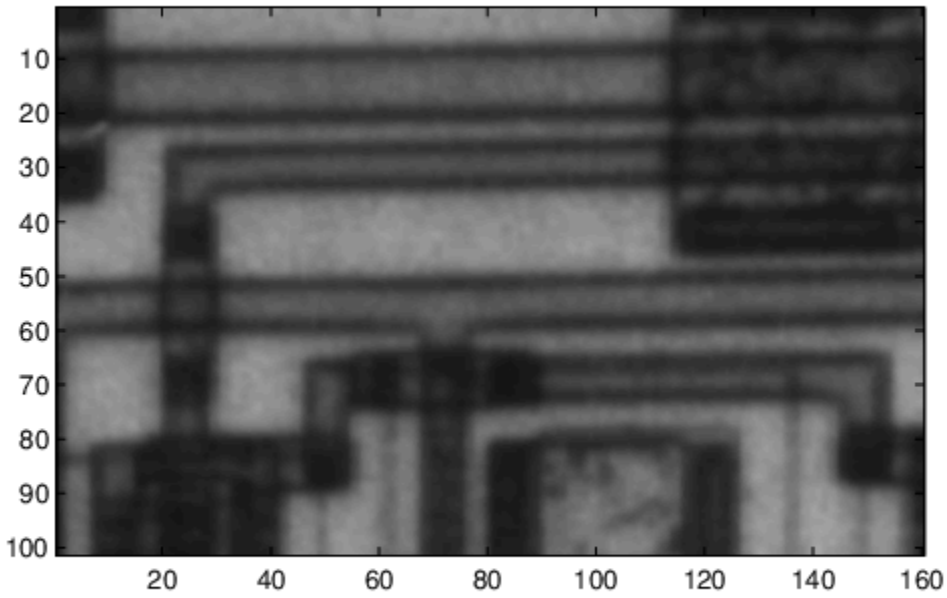
Indexing Operations

Inherited methods perform indexing operations, but return objects of the same class as the subclass.

Therefore, you can index into the image data and call a subclass method:

```
showImage(img1(100:200,1:160));
```

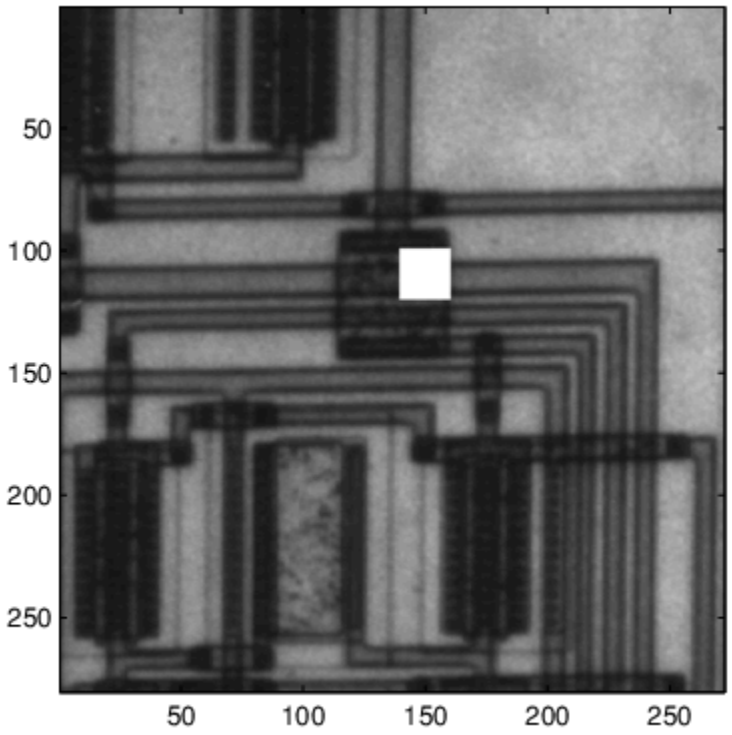
Subscripted reference operations (controlled by the inherited `subsref` method) return a `DocUint8` object.



You can assign values to indexed elements:

```
img1(100:120,140:160) = 255;
img1.showImage;
```

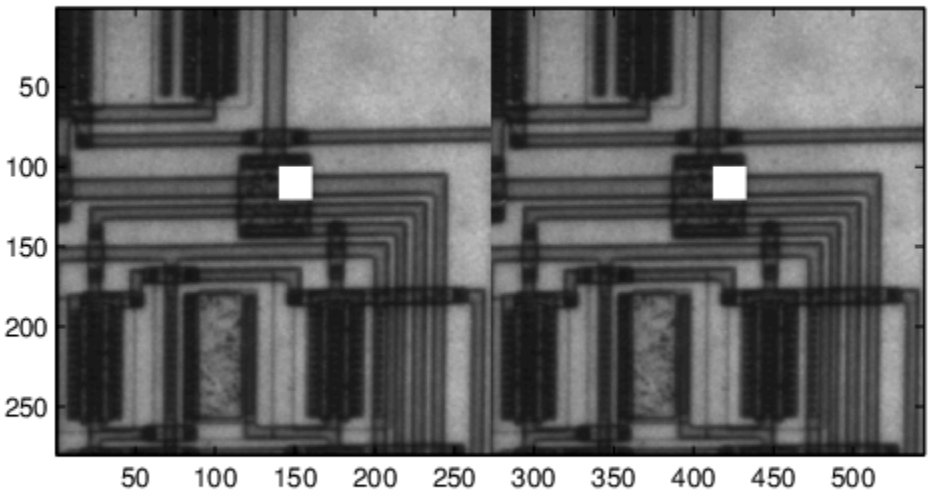
Subscripted assignment operations (controlled by the inherited `subsasgn` method) return a `DocUint8` object.



Concatenation Operations

Concatenation operations work on `DocUint8` objects because this class inherits the `uint8` `horzcat` and `vertcat` methods, which return a `DocUint8` object:

```
showImage([img1 img1]);
```



Data Operations

Methods that operate on data values, such as arithmetic operators, always return an object of the built-in type (not of the subclass type). For example, multiplying `DocUint8` objects returns a `uint8` object, so calling `showImage` throws an error:

```
a = img1.*1.8;
showImage(a);
```

```
Check for missing argument or incorrect argument data type in call to function 'showImage'.
```

To perform operations of this type, implement a subclass method to override the inherited method. The `times` method implements array (element-by-element) multiplication.

Add this method to the `DocUint8` class:

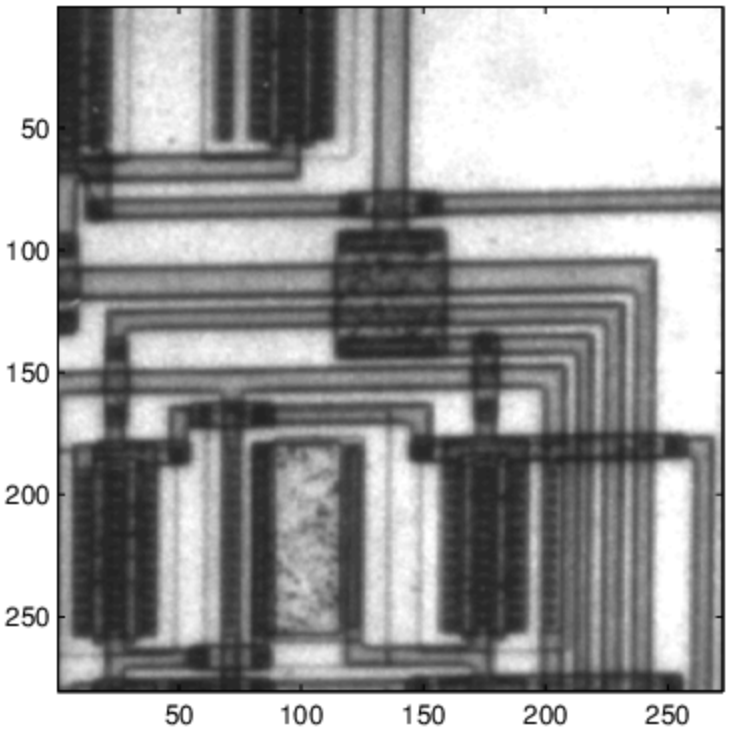
```
function o = times(obj,val)
   u8 = uint8(obj).*val;
   o = DocUint8(u8);
end
```

When you override a `uint8` method, MATLAB calls the subclass method, not the base class method. The subclass method must:

- Call the `uint8` `times` method on the `DocUint8` object data.
- Construct a new `DocUint8` object using the `uint8` data.

After adding the `times` method to the `DocUint8` class, the output of multiplication expressions is an object of the `DocUint8` class:

```
showImage(img1.*1.8);
```



See Also

Related Examples

- “Operator Overloading” on page 17-19
- “Subclasses of Built-In Types with Properties” on page 12-53

Subclasses of Built-In Types with Properties

In this section...

Specialized Numeric Types with Additional Data Storage

Subclass built-in numeric types to create customized data types that inherit the functionality of the built-in type. Add or modify functionality to that provided by the superclass by implementing class methods.

Providing additional data storage in the subclass by defining properties can be a useful extension to the built-in data class. However, the addition of properties to the subclass requires the subclass to define methods to implement standard array behaviors.

For more information, see “Subclasses of MATLAB Built-In Types” on page 12-40.

Subclasses with Properties

When a subclass of a built-in class defines properties, default indexing and concatenation do not work. The default `subsref`, `subsasgn`, `horzcat`, and `vertcat` functions cannot work with unknown property types and values. Therefore, the subclass must define these behaviors by implementing these methods.

This sample implementation of the `ExtendDouble` class derives from the `double` class and defines a single property. The `ExtendDouble` class definition demonstrates how to implement indexing and concatenation for subclasses of built-in classes

Property Added

The `ExtendDouble` class defines the `DataString` property to contain text that describes the data. The superclass part of the class contains the numeric data.

Methods Implemented

The following methods modify the behavior of the `ExtendDouble` class:

- `ExtendDouble` — The constructor supports a no argument syntax that initializes properties to empty values.
- `subsref` — Enables subscripted reference to the superclass part of the subclass, dot notation reference to the `DataString` property, and dot notation reference the built-in data via the name `Data`.

- `subsasgn` — Enables subscripted assignment to the superclass part of the subclass, dot notation reference to the `DataString` property, and dot notation reference the built-in data via the name `Data`.
- `horzcat` — Defines horizontal concatenation of `ExtendDouble` objects. concatenates the superclass part using the `double` class `horzcat` method and forms a cell array of the `DataString` properties.
- `vertcat` — The vertical concatenation equivalent of `horzcat` (both are required).
- `char` — A `ExtendDouble` to `char` converter used by `horzcat` and `vertcat`.
- `disp` — `ExtendDouble` implements a `disp` method to provide a custom display for the object.

Class Definition Code

The `ExtendDouble` class extends `double` and implements methods to support subscripted indexing and concatenation.

```
classdef ExtendDouble < double

   properties
      DataString
   end

   methods
      function obj = ExtendDouble(data,str)
         if nargin == 0
            data = 0;
            str = '';
         elseif nargin == 1
            str = '';
         end
         obj = obj@double(data);
         obj.DataString = str;
      end

      function sref = subsref(obj,s)
         switch s(1).type
            case '.'
               switch s(1).subs
                  case 'DataString'
                     sref = obj.DataString;
                  case 'Data'
                     d = double(obj);
                     if length(s)<2
                        sref = d;
                     elseif length(s)>1 && strcmp(s(2).type,'()')
                        sref = subsref(d,s(2:end));
                     end
                  otherwise
                     error('Not a supported indexing expression')
               end
            case '()'
               d = double(obj);
               newd = subsref(d,s(1:end));
               sref = ExtendDouble(newd,obj.DataString);
            case '{}'
               error('Not a supported indexing expression')
```

```
            end
        end

        function obj = subsasgn(obj,s,b)
            switch s(1).type
                case '.'
                    switch s(1).subs
                        case 'DataString'
                            obj.DataString = b;
                        case 'Data'
                            if length(s)<2
                                obj = ExtendDouble(b,obj.DataString);
                            elseif length(s)>1 && strcmp(s(2).type,'()')
                                d = double(obj);
                                newd = subsasgn(d,s(2:end),b);
                                obj = ExtendDouble(newd,obj.DataString);
                            end
                        otherwise
                            error('Not a supported indexing expression')
                    end
                case '()'
                    d = double(obj);
                    newd = subsasgn(d,s(1),b);
                    obj = ExtendDouble(newd,obj.DataString);
                case '{}'
                    error('Not a supported indexing expression')
            end
        end

        function newobj = horzcat(varargin)
            d1 = cellfun(@double,varargin,'UniformOutput',false );
            data = horzcat(d1{:});
            str = horzcat(cellfun(@char,varargin,'UniformOutput',false));
            newobj = ExtendDouble(data,str);
        end

        function newobj = vertcat(varargin)
            d1 = cellfun(@double,varargin,'UniformOutput',false );
            data = vertcat(d1{:});
            str = vertcat(cellfun(@char,varargin,'UniformOutput',false));
            newobj = ExtendDouble(data,str);
        end

        function str = char(obj)
            str = obj.DataString;
        end

        function disp(obj)
            disp(obj.DataString)
            disp(double(obj))
        end
    end
end
```

Using ExtendDouble

Create an instance of `ExtendDouble` and notice that the display is different from the default:

```
ed = ExtendDouble(1:10,'One to ten')

ed =

One to ten
     1     2     3     4     5     6     7     8     9    10
```

Inherited Methods

The `ExtendDouble` class inherits methods from the class double. To see a list of all public methods defined by the double class, use the `methods` function:

```
methods(double.empty)
```

The `sum` function continues to operate on the superclass part of the object:

```
sum(ed)

ans =
    55
```

The `sort` function works on the superclass part of the object:

```
sort(ed(10:-1:1))

ans =

     1     2     3     4     5     6     7     8     9    10
```

Arithmetic operators work on the superclass part of the object:

```
ed.^2

ans =

     1     4     9    16    25    36    49    64    81   100
```

Subscripted Indexing

Because the `ExtendDouble` class defines a property, the class must implement its own `subsref` and `subsasgn` methods.

This class implements the following subscripted indexing expressions for reference and assignment.

- `obj.DataString` — access the `DataString` property.
- `obj.Data`, `obj.Data(ind)` — access the data using a property-style reference. Reference returns values of type `double`.
- `obj(ind)` — access the numeric data (same as `obj.Data(ind)`). Reference returns values of type `ExtendDouble`.

The class `subsref` method enables you to use `ExtendDouble` objects like numeric arrays to reference the numeric data:

```
ed = ExtendDouble(1:10,'One to ten');
ed(10:-1:1)

ans =
```

```
One to ten
    10     9     8     7     6     5     4     3     2     1
```

Access the numeric data of the `ExtendDouble` using property-style indexing with the arbitrarily chosen name `Data`:

```
ed.Data(10:-1:1)

ans =

One to ten
    10     9     8     7     6     5     4     3     2     1
```

Access the `DataString` property:

```
ed.DataString

ans =

One to ten
```

Subscripted assignment implements similar syntax in the class `subsasgn` method.

```
ed = ExtendDouble(1:10,'One to ten');
ed(11:13) = [11,12,13];
ed.DataString = 'one to thirteen';
ed

ed =

One to thirteen'
     1     2     3     4     5     6     7     8     9    10    11    12    13
```

The `ExtendDouble` inherits converter methods from the double class. For example, MATLAB calls the `char` method to perform this assignment statement.

```
ed(11:13) = ['a','b','c']

ed =

one to thirteen
     1     2     3     4     5     6     7     8     9    10    97    98    99
```

Class of Value Returned by Indexing Expression

The `ExtendDouble` implements two forms of indexed reference in the `subsref` method:

- `obj.Data` and `obj.Data(ind)` — Return values of class `double`
- `obj(ind)` — Return values of class `ExtendDouble`

For example, compare the values returned by these expressions.

```
ed = ExtendDouble(1:10,'One to ten');
a = ed(1)

a =

One to ten
     1

b = ed.Data(1)
```

```
b =

     1

whos

  Name      Size            Bytes  Class           Attributes

  a         1x1               132  ExtendDouble
  b         1x1                 8  double
  ed        1x10              204  ExtendDouble
```

The increased flexibility of the implementation of indexed reference in the `ExtendDouble` class.

Concatenation of ExtendDouble Objects

Create these two objects:

```
ed1 = ExtendDouble([1:10],'One to ten');
ed2 = ExtendDouble([10:-1:1],'Ten to one');
```

Concatenate these objects along the horizontal dimension:

```
hcat = [ed1,ed2]

hcat =

    'One to ten'    'Ten to one'

  Columns 1 through 13

     1     2     3     4     5     6     7     8     9    10    10     9     8

  Columns 14 through 20

     7     6     5     4     3     2     1

whos

  Name      Size            Bytes  Class           Attributes

  ed1       1x10              204  ExtendDouble
  ed2       1x10              204  ExtendDouble
  hcat      1x20              528  ExtendDouble
```

Vertical concatenation works in a similar way:

```
vcat = [ed1;ed2]

vcat =

    'One to ten'    'Ten to one'

     1     2     3     4     5     6     7     8     9    10
    10     9     8     7     6     5     4     3     2     1
```

Both `horzcat` and `vertcat` return a new object of the same class as the subclass.

See Also

Related Examples

- “Subclasses of Built-In Types Without Properties” on page 12-47

Use of size and numel with Classes

In this section...

size and numel

The `size` function returns the dimensions of an array. The `numel` function returns the number of elements in an array, which is equivalent to `prod(size(objArray))`. That is, the product of the array dimensions.

The `size` and `numel` functions work consistently with arrays of user-defined objects. There is generally no need to overload `size` or `numel` in user-defined classes.

Several MATLAB functions use `size` and `numel` to perform their operations. Therefore, if you do overload either of these functions in your class, be sure that objects of your class work as designed with other MATLAB functions.

If your class modifies array indexing, see “Overload numArgumentsFromSubscript Instead of numel” on page 12-64

Built-In Class Behavior

When you use the `size` and `numel` functions in classes derived from built-in classes, these functions behave the same as they behave in the superclass.

Consider the built-in class `double`:

```
d = 1:10;
size(d)

ans =

     1    10

numel(d)

ans =

    10

dsub = d(7:end);
size(dsub)

ans =

     1     4
```

The `double` class defines these behaviors, including parentheses indexing.

Subclasses Inherit Behavior

Unless the subclass explicitly overrides superclass behavior, subclasses behave like their superclasses. For example, `SimpleDouble` subclasses `double` and defines no properties:

```
classdef SimpleDouble < double
   methods
      function obj = SimpleDouble(data)
         if nargin == 0
            data = 0;
         end
         obj = obj@double(data);
      end
   end
end
```

Create an object and assign the values `1:10`:

```
sd = SimpleDouble(1:10);
```

The `size` function returns the size of the superclass part:

```
size(sd)
```

```
ans =

     1    10
```

The `numel` function returns the number of elements in the superclass part:

```
numel(sd)
```

```
ans =

     10
```

Object arrays return the size of the superclass arrays:

```
size([sd;sd])
```

```
ans =

     2    10
```

```
numel([sd;sd])
```

```
ans =

     20
```

The `SimpleDouble` class inherits the indexing behavior of the `double` class:

```
sdsub = sd(7:end);
size(sdsub)
```

```
ans =

     1     4
```

Classes Not Derived from Built-In Classes

Consider a simple value class. This class does not inherit the array-like behaviors of the `double` class. For example:

```
classdef VerySimpleClass
   properties
      Value
   end
end
```

Create an object and assign a 10-element array to the `Value` property:

```
vs = VerySimpleClass;
vs.Value = 1:10;
size(vs)

ans =

     1     1

numel(vs)

ans =

     1

size([vs;vs])

ans =

     2     1

numel([vs;vs])

ans =

     2
```

`vs` is a scalar object. The `Value` property is an array of `doubles`:

```
size(vs.Value)

ans =

     1    10
```

Apply indexing expressions to the object property:

```
vssub = vs.Value(7:end);
size(vssub)

ans =

     1     4
```

The `vs.Value` property is an array of class `double`:

```
class(vs.Value)

ans =

double
```

Create an array of `VerySimpleClass` objects:

```
vsArray(1:10) = VerySimpleClass;
```

The `Value` property for array elements 2 through 10 is empty:

```
isempty([vsArray(2:10).Value])

ans =

     1
```

MATLAB does not apply scalar expansion to object array property value assignment. Use the `deal` function for this purpose:

```
[vsArray.Value] = deal(1:10);
isempty([vsArray.Value])

ans =

     0
```

The deal function assigns values to each `Value` property in the `vsArray` object array.

Indexing rules for object arrays are equivalent to the rules for arrays of `struct`:

```
vsArray(1).Value

ans =

     1     2     3     4     5     6     7     8     9    10

vsArray(1).Value(6)

ans =

     6
```

Change the Behavior of size or numel

Subclasses of built-in numeric classes inherit a `size` method, which operates on the superclass part of the subclass object (these methods are hidden). If you want `size` or `numel` to behave differently, override them by defining a `size` or `numel` method in your subclass.

Other MATLAB functions use the values returned by these functions. If you change the way that `size` and `numel` behave, ensure that the values returned make sense for the intended use of your class.

Overload numArgumentsFromSubscript Instead of numel

If classes implement a `numArgumentsFromSubscript` method, MATLAB calls it instead of `numel` to determine the number of elements returned by indexed expressions that return comma-separated lists. For example, expressions such as:

```
A(1:2).Prop
```

Both `subsref` and `subsasgn` use `numArgumentsFromSubscript`:

- `subsref` — `numArgumentsFromSubscript` computes the number of expected outputs (`nargout`) returned `subsref`.
- `subsasgn` — `numArgumentsFromSubscript` computes the number of expected inputs (`nargin`) that MATLAB assigns as a result of a call to `subsasgn`.

Subclasses of built-in classes always return scalar objects as a result of subscripted reference and always use scalar objects for subscripted assignment.

If you define a class in which `nargout` for `subsref` or `nargin` for `subsasgn` must be a specific value, then overload `numArgumentsFromSubscript` to return that value.

See Also

`numArgumentsFromSubscript`

Determine Array Class

In this section...

Query the Class Name

To determine the class of an array, use the `class` function:

```
a = [2,5,7,11];
class(a)

ans =
double

str = 'Character array';
class(str)

ans =
char
```

Test for Array Class

The `isa` function enables you to test for a specific class or a category of numeric class (`numeric`, `float`, `integer`):

```
a = [2,5,7,11];
isa(a,'double')

ans =
     1
```

Floating-point values (single and double precision values):

```
isa(a,'float')

ans =
     1
```

Numeric values (floating-point and integer values):

```
isa(a,'numeric')

ans =
     1
```

isa Returns True for Subclasses

`isa` returns true for classes derived from the specified class. For example, the `SubInt` class derives from the built-in type `int16`:

```
classdef SubInt < int16
   methods
      function obj = SubInt(data)
         if nargin == 0
            data = 0;
         end
         obj = obj@int16(data);
      end
   end
end
```

By definition, an instance of the `SubInt` class is also an instance of the `int16` class:

```
aInt = SubInt;
isa(aInt,'int16')

ans =
     1
```

Using the `integer` category also returns `true`:

```
isa(aInt,'integer')

ans =
     1
```

Test for Specific Types

The `class` function returns the name of the *most derived* class of an object:

```
class(aInt)

ans =
SubInt
```

Use the `strcmp` function with the `class` function to check for a specific class of an object:

```
a = int16(7);
strcmp(class(a),'int16')

ans =
     1
```

Because the `class` function returns the class name as a `char` vector, the inheritance does not affect the result of the comparison performed by `strcmp`:

```
aInt = SubInt;
strcmp(class(aInt),'int16')

ans =
     0
```

Test for Most Derived Class

If you define functions that require inputs that are:

- MATLAB built-in types

- Not subclasses of MATLAB built-in types

Use the following techniques to exclude subclasses of built-in types from the input arguments.

- Define a cell array that contains the names of built-in types accepted by your function.
- Call `class` and `strcmp` to test for specific types in a MATLAB control statement.

Test an input argument:

```
if strcmp(class(inputArg),'single')
   % Call function
else
   inputArg = single(inputArg);
end
```

Test for Category of Types

Suppose that you create a MEX-function, `myMexFcn`, that requires two numeric inputs that must be of type `double` or `single`:

```
outArray = myMexFcn(a,b)
```

Define a cell array that contains the character arrays `double` and `single`:

```
floatTypes = {'double','single'};

% Test for proper types
if any(strcmp(class(a),floatTypes)) && ...
   any(strcmp(class(b),floatTypes))
   outArray = myMexFcn(a,b);
else
   % Try to convert inputs to avoid error
   ...
end
```

Another Test for Built-In Types

Use `isobject` to separate built-in types from subclasses of built-in types. The `isobject` function returns `false` for instances of built-in types:

```
% Create a int16 array
a = int16([2,5,7,11]);
isobject(a)

ans =
     0
```

Determine if an array is one of the built-in integer types:

```
if isa(a,'integer') && ~isobject(a)
   % a is a built-in integer type
   ...
end
```

Abstract Classes and Class Members

In this section...

Abstract Classes

Abstract classes are useful for describing functionality that is common to a group of classes, but requires unique implementations within each class.

Abstract Class Terminology

abstract class — A class that cannot be instantiated, but that defines class components used by subclasses.

abstract members — Properties or methods declared in an abstract class, but implemented in subclasses.

concrete class — A class that can be instantiated. Concrete classes contain no abstract members.

concrete members — Properties or methods that are fully implemented by a class.

interface — An abstract class describing functionality that is common to a group of classes, but that requires unique implementations within each class. The abstract class defines the interface of each subclass without specifying the actual implementation.

An abstract class serves as a basis (that is, a superclass) for a group of related subclasses. An abstract class can define abstract properties and methods that subclasses implement. Each subclass can implement the concrete properties and methods in a way that supports their specific requirements.

Implementing a Concrete Subclass

A subclass must implement all inherited abstract properties and methods to become a concrete class. Otherwise, the subclass is itself an abstract class.

MATLAB does not force subclasses to implement concrete methods with the same signature or attributes.

Abstract classes:

- Can define properties and methods that are not abstract
- Pass on their concrete members through inheritance
- Do not need to define any abstract members

Declare Classes as Abstract

A class is abstract when it declares:

- The `Abstract` class attribute
- An abstract method
- An abstract property

If a subclass of an abstract class does not define concrete implementations for all inherited abstract methods or properties, it is also abstract.

Abstract Class

Declare a class as abstract in the `classdef` statement:

```
classdef (Abstract) AbsClass
   ...
end
```

For classes that declare the `Abstract` class attribute:

- Concrete subclasses must redefine any properties or methods that are declared as abstract.
- The abstract class does not need to define any abstract methods or properties.

When you define any abstract methods or properties, MATLAB automatically sets the class `Abstract` attribute to `true`.

Abstract Methods

Define an abstract method:

```
methods (Abstract)
   abstMethod(obj)
end
```

For methods that declare the `Abstract` method attribute:

- Do not use a `function...end` block to define an abstract method, use only the method signature.
- Abstract methods have no implementation in the abstract class.
- Concrete subclasses are not required to support the same number of input and output arguments and do not need to use the same argument names. However, subclasses generally use the same signature when implementing their version of the method.
- Abstract methods cannot define `arguments` blocks.

Abstract Properties

Define an abstract property:

```
properties (Abstract)
   AbsProp
end
```

For properties that declare the `Abstract` property attribute:

- Concrete subclasses must redefine abstract properties without the `Abstract` attribute.
- Concrete subclasses must use the same values for the `SetAccess` and `GetAccess` attributes as those attributes used in the abstract superclass.

- Abstract properties cannot define access methods and cannot specify initial values. The subclass that defines the concrete property can create access methods and specify initial values.

For more information on access methods, see “Property Get and Set Methods” on page 8-38.

Determine If a Class Is Abstract

Determine if a class is abstract by querying the `Abstract` property of its `meta.class` object. For example, the `AbsClass` defines two abstract methods:

```
classdef AbsClass
   methods(Abstract)
      result = absMethodOne(obj)
      output = absMethodTwo(obj)
   end
end
```

Use the logical value of the `meta.class` `Abstract` property to determine if the class is abstract:

```
mc = ?AbsClass;
if ~mc.Abstract
   % not an abstract class
end
```

Display Abstract Member Names

Use the `meta.abstractDetails` function to display the names of abstract properties or methods and the names of the defining classes:

```
meta.abstractDetails('AbsClass');
```

```
Abstract methods for class AbsClass:
   absMethodTwo   % defined in AbsClass
   absMethodOne   % defined in AbsClass
```

Find Inherited Abstract Properties and Methods

The `meta.abstractDetails` function returns the names and defining class of any inherited abstract properties or methods that you have not implemented in your subclass. Use this function if you want the subclass to be concrete and must determine what abstract members the subclass inherits.

For example, suppose that you create a subclass of the `AbsClass` class that is defined in the previous section. In this case, the subclass implements only one of the abstract methods defined by `AbsClass`.

```
classdef SubAbsClass < AbsClass
% Does not implement absMethodOne
% defined as abstract in AbsClass
   methods
      function out = absMethodTwo(obj)
         ...
      end
   end
end
```

Determine if you implemented all inherited class members using `meta.abstractDetails`:

```
meta.abstractDetails(?SubAbsClass)

Abstract methods for class SubAbsClass:
   absMethodOne   % defined in AbsClass
```

The `SubAbsClass` class is abstract because it has not implemented the `absMethodOne` method defined in `AbsClass`.

```
msub = ?SubAbsClass;
msub.Abstract

ans =

     1
```

If you implement both methods defined in `AbsClass`, the subclass becomes concrete.

See Also

Related Examples

- “Define an Interface Superclass” on page 12-72

Define an Interface Superclass

In this section...

Interfaces

The properties and methods defined by a class form the interface that determines how class users interact with objects of the class. When creating a group of related classes, interfaces define a common interface to all these classes. The actual implementations of the interface can differ from one class to another.

Consider a set of classes designed to represent various types of graphs. All classes must implement a `Data` property to contain the data used to generate the graph. However, the form of the data can differ considerably from one type of graph to another. Each class can implement the `Data` property differently.

The same differences apply to methods. All classes can have a `draw` method that creates the graph, but the implementation of this method changes with the type of graph.

The basic idea of an interface class is to specify the properties and methods that each subclass must implement without defining the actual implementation. This approach enables you to enforce a consistent interface to a group of related objects. As you add more classes in the future, the interface remains the same.

Interface Class Implementing Graphs

This example creates an interface for classes used to represent specialized graphs. The interface is an abstract class that defines properties and methods that the subclasses must implement, but does not specify how to implement these components.

This approach enforces the use of a consistent interface while providing the necessary flexibility to implement the internal workings of each specialized subclass differently.

In this example, a package folder contains the interface, derived subclasses, and a utility function:

```
+graphics/GraphInterface.m  % abstract interface class
+graphics/LineGraph.m       % concrete subclass
```

Interface Properties and Methods

The `graph` class specifies the following properties, which the subclasses must define:

- `Primitive` — Handle of the graphics object used to implement the specialized graph. The class user has no need to access these objects directly so this property has `protected` `SetAccess` and `GetAccess`.
- `AxesHandle` — Handle of the axes used for the graph. The specialized `graph` objects can set axes object properties. This property has `protected` `SetAccess` and `GetAccess`.
- `Data` — All subclasses of the `GraphInterface` class must store data. The type of data varies and each subclass defines the storage mechanism. Subclass users can change the data values so this property has public access rights.

The `GraphInterface` class names three abstract methods that subclasses must implement. The `GraphInterface` class also suggests in comments that each subclass constructor must accept the plot data and property name/property value pairs for all class properties.

- Subclass constructor — Accept data and P/V pairs and return an object.
- `draw` — Used to create a drawing primitive and render a graph of the data according to the type of graph implemented by the subclass.
- `zoom` — Implementation of a zoom method by changing the axes `CameraViewAngle` property. The interface suggests the use of the `camzoom` function for consistency among subclasses. The zoom buttons created by the `addButtons` static method use this method as a callback.
- `updateGraph` — Method called by the `set.Data` method to update the plotted data whenever the `Data` property changes.

Interface Guides Class Design

The package of classes that derive from the `GraphInterface` abstract class implement the following behaviors:

- Creating an instance of a specialized `GraphInterface` object (subclass object) without rendering the plot
- Specifying any or none of the object properties when you create a specialized `GraphInterface` object
- Changing any object property automatically updates the currently displayed plot
- Allowing each specialized `GraphInterface` object to implement whatever additional properties it requires to give class users control over those characteristics.

Define the Interface

The `GraphInterface` class is an abstract class that defines the methods and properties used by the subclasses. Comments in the abstract class describe the intended implementation:

```
classdef GraphInterface < handle
   % Abstract class for creating data graphs
   % Subclass constructor should accept
   % the data that is to be plotted and
   % property name/property value pairs
   properties (SetAccess = protected, GetAccess = protected)
      Primitive
      AxesHandle
   end
   properties
      Data
   end
   methods (Abstract)
      draw(obj)
      % Use a line, surface,
      % or patch graphics primitive
      zoom(obj,factor)
      % Change the CameraViewAngle
      % for 2D and 3D views
      % use camzoom for consistency
      updateGraph(obj)
      % Update the Data property and
      % update the drawing primitive
```

```
    end

    methods
        function set.Data(obj,newdata)
            obj.Data = newdata;
            updateGraph(obj)
        end
        function addButtons(gobj)
            hfig = get(gobj.AxesHandle,'Parent');
            uicontrol(hfig,'Style','pushbutton','String','Zoom Out',...
                'Callback',@(src,evnt)zoom(gobj,.5));
            uicontrol(hfig,'Style','pushbutton','String','Zoom In',...
                'Callback',@(src,evnt)zoom(gobj,2),...
                'Position',[100 20 60 20]);
        end
    end
end
```

The `GraphInterface` class implements the property set method (`set.Data`) to monitor changes to the `Data` property. An alternative is to define the `Data` property as `Abstract` and enable the subclasses to determine whether to implement a set access method for this property. The `GraphInterface` class defines a set access method that calls an abstract method (`updateGraph`, which each subclass must implement). The `GraphInterface` interface imposes a specific design on the whole package of classes, without limiting flexibility.

Method to Work with All Subclasses

The `addButtons` method adds push buttons for the `zoom` methods, which each subclass must implement. Using a method instead of an ordinary function enables `addButtons` to access the protected class data (the axes handle). Use the object `zoom` method as the push-button callback.

```
function addButtons(gobj)
    hfig = get(gobj.AxesHandle,'Parent');
    uicontrol(hfig,'Style','pushbutton',...
        'String','Zoom Out',...
        'Callback',@(src,evnt)zoom(gobj,.5));
    uicontrol(hfig,'Style','pushbutton',...
        'String','Zoom In',...
        'Callback',@(src,evnt)zoom(gobj,2),...
        'Position',[100 20 60 20]);
end
```

Derive a Concrete Class — LineGraph

This example defines only a single subclass used to represent a simple line graph. It derives from `GraphInterface`, but provides implementations for the abstract methods `draw`, `zoom`, `updateGraph`, and its own constructor. The base class `GraphInterface` and subclass are all contained in a package (`graphics`), which you must use to reference the class name:

```
classdef LineGraph < graphics.GraphInterface
```

Add Properties

The `LineGraph` class implements the interface defined in the `GraphInterface` class and adds two additional properties—`LineColor` and `LineType`. This class defines initial values for each property, so specifying property values in the constructor is optional. You can create a `LineGraph` object with no data, but you cannot produce a graph from that object.

```
properties
   LineColor = [0 0 0];
   LineType = '-';
end
```

The LineGraph Constructor

The constructor accepts a `struct` with `x` and `y` coordinate data, and property name/property value pairs:

```
function gobj = LineGraph(data,varargin)
   if nargin > 0
      gobj.Data = data;
      if nargin > 2
         for k=1:2:length(varargin)
            gobj.(varargin{k}) = varargin{k+1};
         end
      end
   end
end
```

Implement the draw Method

The `LineGraph` `draw` method uses property values to create a `line` object. The `LineGraph` class stores the `line` handle as protected class data. To support the use of no input arguments for the class constructor, `draw` checks the `Data` property to determine if it is empty before proceeding:

```
function gobj = draw(gobj)
   if isempty(gobj.Data)
      error('The LineGraph object contains no data')
   end
   h = line(gobj.Data.x,gobj.Data.y,...
      'Color',gobj.LineColor,...
      'LineStyle',gobj.LineType);
   gobj.Primitive = h;
   gobj.AxesHandle = get(h,'Parent');
end
```

Implement the zoom Method

The `LineGraph` `zoom` method follows the comments in the `GraphInterface` class which suggest using the `camzoom` function. `camzoom` provides a convenient interface to zooming and operates correctly with the push buttons created by the `addButtons` method.

Define the Property Set Methods

Property set methods provide a convenient way to execute code automatically when the value of a property changes for the first time in a constructor. (See “Property Get and Set Methods” on page 8-38.) The `linegraph` class uses set methods to update the `line` primitive data (which causes a redraw of the plot) whenever a property value changes. The use of property set methods provides a way to update the data plot quickly without requiring a call to the `draw` method. The `draw` method updates the plot by resetting all values to match the current property values.

Three properties use set methods: `LineColor`, `LineType`, and `Data`. `LineColor` and `LineType` are properties added by the `LineGraph` class and are specific to the `line` primitive used by this class. Other subclasses can define different properties unique to their specialization (for example, `FaceColor`).

The `GraphInterface` class implements the `Data` property set method. However, the `GraphInterface` class requires each subclass to define a method called `updateGraph`, which handles the update of plot data for the specific drawing primitive used.

The LineGraph Class

Here is the `LineGraph` class definition.

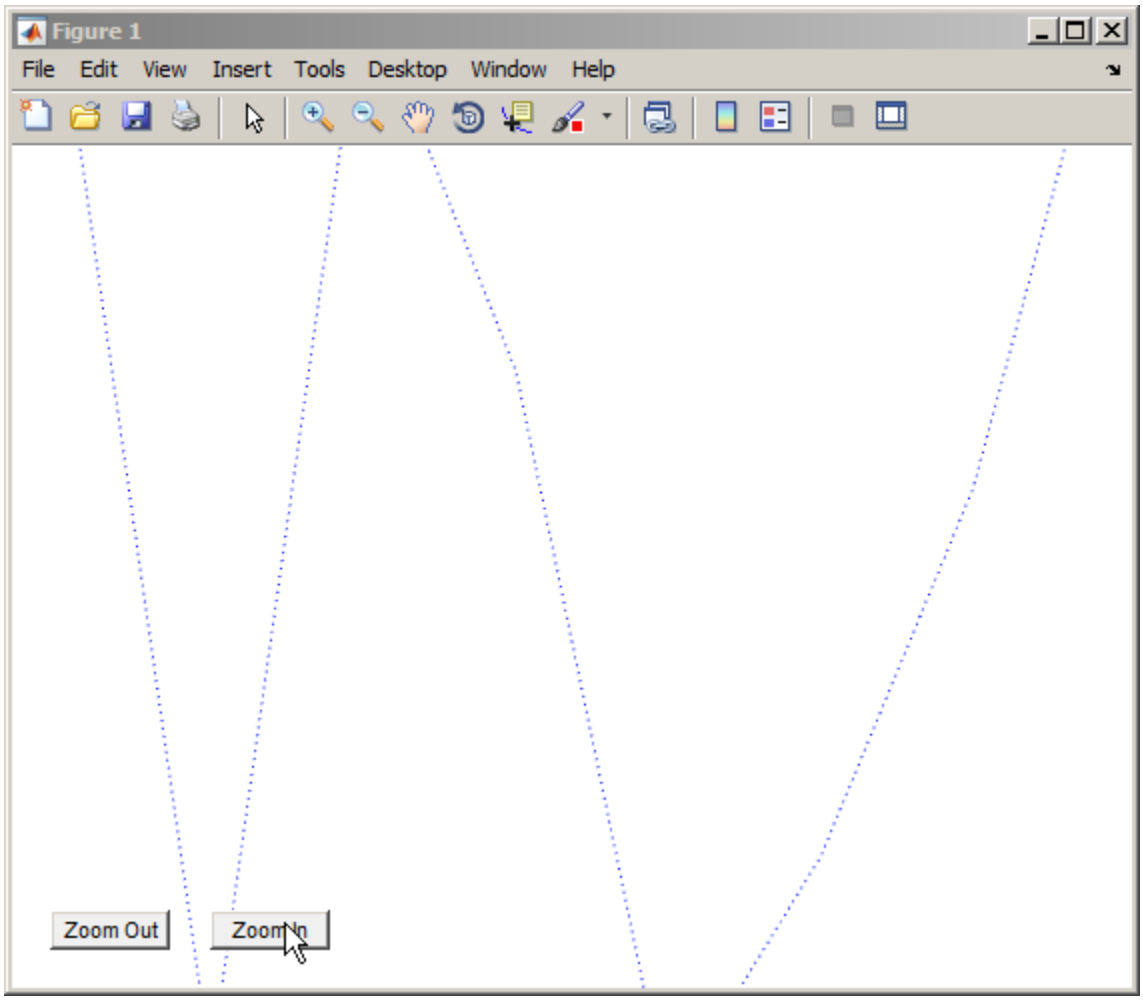
```
classdef LineGraph < graphics.GraphInterface
   properties
      LineColor = [0 0 0]
      LineType = '-'
   end

   methods
      function gobj = LineGraph(data,varargin)
         if nargin > 0
            gobj.Data = data;
            if nargin > 1
               for k=1:2:length(varargin)
                  gobj.(varargin{k}) = varargin{k+1};
               end
            end
         end
      end

      function gobj = draw(gobj)
         if isempty(gobj.Data)
            error('The LineGraph object contains no data')
         end
         h = line(gobj.Data.x,gobj.Data.y,...
            'Color',gobj.LineColor,...
            'LineStyle',gobj.LineType);
         gobj.Primitive = h;
         gobj.AxesHandle = h.Parent;
      end

      function zoom(gobj,factor)
         camzoom(gobj.AxesHandle,factor)
      end

      function updateGraph(gobj)
         set(gobj.Primitive,...
            'XData',gobj.Data.x,...
            'YData',gobj.Data.y)
      end

      function set.LineColor(gobj,color)
         gobj.LineColor = color;
         set(gobj.Primitive,'Color',color)
      end

      function set.LineType(gobj,ls)
         gobj.LineType = ls;
         set(gobj.Primitive,'LineStyle',ls)
      end
   end
end
```

Use the LineGraph Class

The `LineGraph` class defines the simple API specified by the `graph` base class and implements its specialized type of graph:

```
d.x = 1:10;
d.y = rand(10,1);
lg = graphics.LineGraph(d,'LineColor','b','LineType',':');
lg.draw;
lg.addButtons;
```

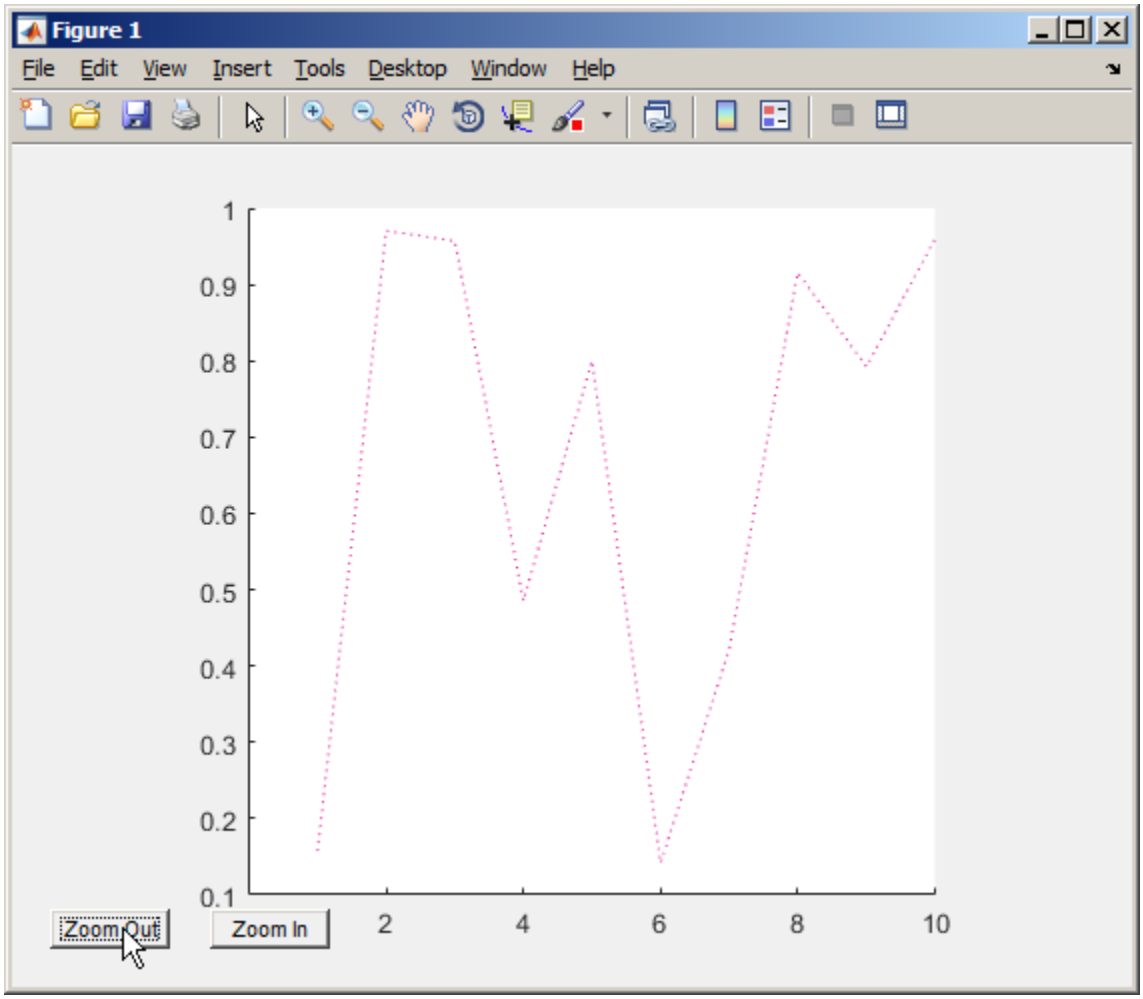
Clicking the **Zoom In** button shows the `zoom` method providing the callback for the button.



Changing properties updates the graph:

```
d.y = rand(10,1);
lg.Data = d;
lg.LineColor = [0.9,0.1,0.6];
```

Now click **Zoom Out** and see the new results:



See Also

Related Examples

- “Abstract Classes and Class Members” on page 12-68

Saving and Loading Objects

Save and Load Process for Objects

In this section...

Save and Load Objects

Use `save` and `load` to store and reload objects:

```
save filename object
load filename object
```

What Information Is Saved?

Saving objects in MAT-files saves:

- The full name of the object class, including any package qualifiers
- Values of dynamic properties
- All property default values defined by the class at the time the first object of the class is saved to the MAT-file.
- The names and values of all properties, with the following exceptions:
  - Properties are not saved if their current values are the same as the default values specified in the class definition.
  - Properties are not saved if their `Transient`, `Constant`, or `Dependent` attributes set to `true`.

For a description of property attributes, see “Property Attributes” on page 8-8.

To save graphics objects, see `savefig`.

How Is the Property Data Loaded?

When loading objects from MAT-files, the `load` function restores the object.

- `load` creates a new object.
- If the class `ConstructOnLoad` attribute is set to `true`, `load` calls the class constructor with no arguments. Otherwise, `load` does not call the class constructor.
- `load` assigns the saved property values to the object properties. These assigned values are subjected to any property validation defined by the class. Then any property set methods defined by the class are called, (except in the case of `Dependent`, `Constant`, or `Transient` properties, which are not saved or loaded).
- `load` assigns the default values saved in the MAT-file to properties whose values were not saved because the properties were set to the default values when saved. These assignments result in calls to property set methods defined by the class.

- If a property of an object being loaded contains an object, then `load` creates a new object of the same class and assigns it to the property. If the object contained in the property is a handle object, then the property contains a new handle object of the same class.

MATLAB calls property set methods to ensure that property values are still valid in cases where the class definition has changed.

For information, see “Property Get and Set Methods” on page 8-38 and “Validate Property Values” on page 8-18.

Errors During Load

If a new version of a class removes, renames, or changes the validation for a property, `load` can generate an error when attempting to set the altered or deleted property.

When an error occurs while an object is being loaded from a file, MATLAB does one of the following:

- If the class defines a `loadobj` method, MATLAB returns the saved values to the `loadobj` method in a `struct`.
- If the class does not define a `loadobj` method, MATLAB silently ignores the errors. The `load` function reconstitutes the object with property values that do not produce an error.

In the `struct` passed to the `loadobj` method, the field names correspond to the property names. The field values are the saved values for the corresponding properties.

If the saved object derives from multiple superclasses that have private properties with same name, the `struct` contains only the property value of the most direct superclass.

For information on how to implement `saveobj` and `loadobj` methods, see “Modify the Save and Load Process” on page 13-12.

Changes to Property Validation

If a class definition changes property validation such that loaded property values are no longer valid, MATLAB substitutes the currently defined default value for that property. The class can define a `loadobj` method or converter methods to provide compatibility among class versions.

For information on property validation, see “Validate Property Values” on page 8-18

See Also

`isequal`

Related Examples

- “Object Save and Load”

Reduce MAT-File Size for Saved Objects

In this section...

Default Values

If a property often has the same value, define a default value for that property. When the user saves the object to a MAT-file, MATLAB does not save the value of a property if the current value equals the default value. MATLAB saves the default value on a per class basis to avoid saving the value for every object.

For more information on how MATLAB evaluates default value expressions, see “Define Properties with Default Values” on page 8-13.

Dependent Properties

Use a dependent property when the property value must be calculated at run time. A dependent property is not saved in the MAT-file when you save an object. Instances of the class do not allocate memory to hold a value for a dependent property.

`Dependent` is a property attribute (see “Property Attributes” on page 8-8 for a complete list.)

Transient Properties

MATLAB does not store the values of transient properties. Transient properties can store data in the object temporarily as an intermediate computation step or for faster retrieval. Use transient properties when you easily can reproduce the data at run time or when the data represents intermediate state that can be discarded.

Avoid Saving Unwanted Variables

Do not save variables that you do not want to load. Be sure that an object is still valid before you save it. For example, if you save a deleted handle object, MATLAB loads it as a deleted handle.

See Also

Related Examples

- “Modify the Save and Load Process” on page 13-12
- “Object Save and Load”

Save Object Data to Recreate Graphics Objects

In this section...

What to Save

Use transient properties to avoid saving what you can recreate when loading the object. For example, an object can contain component parts that you can regenerate from data that is saved. Regenerating these components also enables newer versions of the class to create the components in a different way.

Regenerate When Loading

The `YearlyRainfall` class illustrates how to regenerate a graph when loading objects of that class. `YearlyRainfall` objects contain a bar chart of the monthly rainfall for a given location and year. The `Location` and `Year` properties are ordinary properties whose values are saved when you save the object.

The `Chart` property contains the handle to the bar chart. When you save a bar chart, MATLAB also saves the figure, axes, and Bar object and the data required to create these graphics objects. The `YearlyRainfall` class design eliminates the need to save objects that it can regenerate:

- The `Chart` property is `Transient` so the graphics objects are not saved.
- `ChartData` is a private property that provides storage for the Bar object data (`YData`).
- The `load` function calls the `set.ChartData` method, passing it the saved bar chart data.
- The `setup` method regenerates the bar chart and assigns the handle to the `Chart` property. Both the class constructor and the `set.ChartData` method call `setup`.

```
classdef YearlyRainfall < handle
   properties
      Location
      Year
   end
   properties(Transient)
      Chart
   end
   properties(Access = private)
      ChartData
   end
   methods
      function rf = YearlyRainfall(data)
         setup(rf,data);
      end
      function set.ChartData(obj,V)
         setup(obj,V);
      end
      function V = get.ChartData(obj)
         V = obj.Chart.YData;
```

```
      end
   end
   methods(Access = private)
      function setup(rf,data)
         rf.Chart = bar(data);
      end
   end
end
```

Change to a Stairstep Chart

An advantage of the `YearlyRainfall` class design is the flexibility to modify the type of graph used without making previously saved objects incompatible. Loading the object recreates the graph based only on the data that is saved to the MAT-file.

For example, change the type of graph from a bar chart to a stair-step graph by modifying the `setup` method:

```
methods(Access = private)
   function setup(rf,data)
      rf.Chart = stairs(data);
   end
end
```

See Also

Related Examples

- “Modify the Save and Load Process” on page 13-12
- “Object Save and Load”

Improve Version Compatibility with Default Values

In this section...

Version Compatibility

Default property values can help you implement version compatibility for saved objects. For example, suppose that you add a property to version 2 of your class. Having a default value enables MATLAB to assign a value to the new property when loading a version 1 object.

Similarly, suppose version 2 of your class removes a property. If a version 2 object is saved and loaded into version 1, your `loadobj` method can use the default value from version 1.

Using a Default Property Value

The `EmployeeInfo` class shows how to use property default values as a way to enhance compatibility among versions. Version 1 of the `EmployeeInfo` class defines three properties — `Name`, `JobTitle`, and `Department`.

```
classdef EmployeeInfo
   properties
      Name
      JobTitle
      Department
   end
end
```

Version 2 of the `EmployeeInfo` class adds a property, `Country`, for the country name of the employee location. The `Country` property has a default value of `'USA'`.

```
classdef EmployeeInfo
   properties
      Name
      JobTitle
      Department
      Country = 'USA'
   end
end
```

The character array, `'USA'`, is a good default value because:

- MATLAB assigns an empty double `[]` to properties that do not have default values defined by the class. Empty double is not a valid value for the `Country` property.
- In version 1, all employees were in the USA. Therefore, any version 1 object loaded into version 2 receives a valid value for the `Country` property.

See Also

Related Examples

- “Modify the Save and Load Process” on page 13-12
- “Object Save and Load”

Avoid Property Initialization Order Dependency

In this section...

Control Property Loading

Problems can occur if property values depend on the order in which `load` sets the property values.

Suppose that your class design is such that both of the following are true:

- A property set method changes another property value.
- A property value is computed from other property values.

Then the final state of an object after changing a series of property values can depend on the order in which you set the properties. This order dependency can affect the result of loading an object.

The `load` function sets property values in a particular order. This order can be different from the order in which you set the properties in the saved object. As a result, the loaded object can have different property values than the object had when it was saved.

Restore Nondependent Properties

If a property set function changes the values of other properties, then define the `Dependent` attribute of that property as `true`. MATLAB does not save or restore dependent property values.

Use nondependent properties for storing the values set by the dependent property. Then the `load` function restores the nondependent properties with the same values that were saved. The `load` function does not call the dependent property set method because there is no value in the saved file for that property.

Dependent Property with Private Storage

The `Odometer` class avoids order dependences when loading objects by controlling which properties are restored when loading:

- The `Units` property is dependent. Its property set method sets the `TotalDistance` property. Therefore `load` does not call the `Units` property set method.
- The load function restores `TotalDistance` to whatever value it had when you saved the object.

```
classdef Odometer
   properties(Constant)
      ConversionFactor = 1.6
   end
   properties
      TotalDistance = 0
   end
```

```
    properties(Dependent)
       Units
    end
    properties(Access=private)
       PrivateUnits = 'mi'
    end
    methods
       function unit = get.Units(obj)
          unit = obj.PrivateUnits;
       end
       function obj = set.Units(obj,newUnits)
          % validate newUnits to be a char vector
          switch(newUnits)
             case 'mi'
                if strcmp(obj.PrivateUnits,'km')
                   obj.TotalDistance = obj.TotalDistance / ...
                      obj.ConversionFactor;
                   obj.PrivateUnits = newUnits;
                end
             case 'km'
                if strcmp(obj.PrivateUnits,'mi')
                   obj.TotalDistance = obj.TotalDistance * ...
                      obj.ConversionFactor;
                   obj.PrivateUnits = newUnits;
                end
             otherwise
                error('Odometer:InvalidUnits', ...
                   'Units ''%s'' is not supported.', newUnits);
          end
       end
    end
end
```

Suppose that you create an instance of `Odometer` and set the following property values:

```
odObj = Odometer;
odObj.Units = 'km';
odObj.TotalDistance = 16;
```

When you save the object:

- `ConversionFactor` is not saved because it is a `Constant` property.
- `TotalDistance` is saved.
- `Units` is not saved because it is a `Dependent` property.
- `PrivateUnits` is saved and provides the storage for the current value of `Units`.

When you load the object:

- `ConversionFactor` is obtained from the class definition.
- `TotalDistance` is loaded.
- `Units` is not loaded, so its set method is not called.
- `PrivateUnits` is loaded from the saved object.

If the `Units` property was not `Dependent`, loading it calls its set method and causes the `TotalDistance` property to be set again.

Property Value Computed from Other Properties

The `Odometer2` class `TripDistance` property depends only on the values of two other properties, `TotalDistance` and `TripMarker`.

The class avoids order dependence when initializing property values during the load process by making the `TripDistance` property dependent. MATLAB does not save or load a value for the `TripDistance` property, but does save and load values for the two properties used to calculate `TripDistance` in its property get method.

```
classdef Odometer2
   properties
      TotalDistance = 0
      TripMarker = 0
   end
   properties(Dependent)
      TripDistance
   end
   methods
      function distance = get.TripDistance(obj)
         distance = obj.TotalDistance - obj.TripMarker;
      end
   end
end
```

See Also

Related Examples

- “Modify the Save and Load Process” on page 13-12
- “Object Save and Load”

Modify the Save and Load Process

In this section...

When to Modify the Save and Load Process

The primary reason for modifying the save and load process is to support backward and forward compatibility of classes. Consider modifying the save and load process when you:

- Rename a class
- Remove properties
- Define a circular reference of handle objects where initialization order is important
- Must call the constructor with arguments and, therefore, cannot use `ConstructOnLoad`

How to Modify the Save and Load Process

The most versatile technique for modifying the save and load process is to implement `loadobj`, and if necessary, `saveobj` methods for your class. MATLAB executes these methods when you call `save` or `load` on an object of the class.

The `save` function calls your class `saveobj` method before performing the save operation. The `save` function then saves the value returned by the `saveobj` method. You can use `saveobj` to return a modified object or a `struct` that contains property values.

`load` calls your class `loadobj` method after loading the object. The `load` function loads the value returned by the `loadobj` method into the workspace. A `loadobj` method can modify the object being loaded or can reconstruct an object from the data saved by the class `saveobj` method.

Implementing saveobj and loadobj Methods

Implement a `saveobj` method that modifies the object being saved, then implement a `loadobj` method to return the object to the correct state when loading it.

Implement the `loadobj` method as a `Static` method because MATLAB can call the `loadobj` method with a `struct` instead of an object of the class.

Implement the `saveobj` method as an ordinary method (that is, calling it requires an instance of the class).

MATLAB saves the object class name so that `load` can determine which `loadobj` method to call in cases where your `saveobj` method saves only the object data in a structure. Therefore, the class must be accessible to MATLAB when you load the object.

Use a `loadobj` method when:

- The class definition has changed since the object was saved, requiring you to modify the object before loading.
- A `saveobj` method modified the object during the save operation, possibly saving data in a `struct`. Implement the `loadobj` method to reconstruct the object from the output of `saveobj`.

Additional Considerations

When you decide to modify the default save and load process, keep the following points in mind:

- If loading any property value from the MAT-file produces an error, `load` passes a `struct` to `loadobj`. The `struct` field names correspond to the property names extracted from the file.
- `loadobj` must always be able to accept a `struct` as input and return an object, even if there is no `saveobj` or `saveobj` does not return a `struct`.
- If `saveobj` returns a `struct`, then `load` always passes that `struct` to `loadobj`.
- Subclass objects inherit superclass `loadobj` and `saveobj` methods. Therefore, if you do not implement a `loadobj` or `saveobj` method in the subclass, MATLAB calls only the inherited methods.

  If a superclass implements a `loadobj` or `saveobj` method, then a subclass can also implement a `loadobj` or `saveobj` method that calls the superclass methods. For more information, see “Save and Load Objects from Class Hierarchies” on page 13-24.
- The `load` function does not call the constructor by default. For more information, see “Initialize Objects When Loading” on page 13-22.

See Also

Related Examples

- “Basic saveobj and loadobj Pattern” on page 13-14
- “Object Save and Load”

Basic saveobj and loadobj Pattern

In this section...

Using saveobj and loadobj

Depending on the requirements of your class, there are various ways you can use `saveobj` and `loadobj` methods. This pattern is a flexible way to solve problems that you cannot address by simpler means.

The basic process is:

- Use `saveobj` to save all essential data in a `struct` and do not save the entire object.
- Use `loadobj` to reconstruct the object from the saved data.

This approach is not useful in cases where you cannot save property values in a `struct` field. Data that you cannot save, such as a file identifier, you can possibly regenerate in the `loadobj` method.

If you implement a `saveobj` method without implementing a `loadobj` method, MATLAB loads a default object of the class using the current class definition. Add a `loadobj` method to the class to create an object using the data saved with the `saveobj` method.

saveobj

For this pattern, define `saveobj` as an ordinary method that accepts an object of the class and returns a `struct`.

- Copy each property value to a structure field of the same name.
- You can save only the data that is necessary to rebuild the object. Avoid saving whole objects hierarchies, such as those created by graphs.

```
methods
   function s = saveobj(obj)
      s.Prop1 = obj.Prop1;
      s.Prop2 = obj.Prop2
      s.Data = obj.GraphHandle.YData;
   end
end
```

loadobj

Define `loadobj` as a static method. Create an object by calling the class constructor. Then assign values to properties from the `struct` passed to `loadobj`. Use the data to regenerate properties that were not saved.

```
methods(Static)
   function obj = loadobj(s)
      if isstruct(s)
         newObj = ClassConstructor;
         newObj.Prop1 = s.Prop1;
         newObj.Prop2 = s.Prop2
```

```
            newObj.GraphHandle = plot(s.Data);
            obj = newObj;
        else
            obj = s;
        end
    end
end
```

If the `load` function encounters an error, `load` passes `loadobj` a `struct` instead of an object. Your `loadobj` method must always be able to handle a `struct` as the input argument. The input to `loadobj` is always a scalar.

Handle Load Problems

`loadobj` can handle a `struct` input even if you are not using a `saveobj` method.

The `GraphExpression` class creates a graph of a MATLAB expression over a specified range of data. `GraphExpression` uses its `loadobj` method to regenerate the graph, which is not saved with the object.

```
classdef GraphExpression
    properties
        FuncHandle
        Range
    end
    methods
        function obj = GraphExpression(fh,rg)
            obj.FuncHandle = fh;
            obj.Range = rg;
            makeGraph(obj)
        end
        function makeGraph(obj)
            rg = obj.Range;
            x = min(rg):max(rg);
            data = obj.FuncHandle(x);
            plot(data)
        end
    end
    methods (Static)
        function obj = loadobj(s)
            if isstruct(s)
                fh = s.FuncHandle;
                rg = s.Range;
                obj = GraphExpression(fh,rg);
            else
                makeGraph(s);
                obj = s;
            end
        end
    end
end
```

Save and Load Object

Create an object with an anonymous function and a range of data as inputs:

```
h = GraphExpression(@(x)x.^4,[1:25])
```

```
h =

  GraphExpression with properties:

    FuncHandle: @(x)x.^4
         Range: [1x25 double]
```

Save the `GraphExpression` object and close the graph:

```
save myFile h
close
```

Load the object. MATLAB recreates the graph:

```
load myFile h
```

If the `load` function cannot create the object and passes a `struct` to `loadobj`, `loadobj` attempts to create an object with the data supplied.

See Also

Related Examples

- “Modify the Save and Load Process” on page 13-12
- “Object Save and Load”

Maintain Class Compatibility

In this section...

Rename Property

Suppose that you want to rename a property, but do not want to cause errors in existing code that refer to the original property. For example, rename a public property called `OfficeNumber` to `Location`. Here is the original class definition:

```
classdef EmployeeList
   properties
      Name
      Email
      OfficeNumber % Rename as Location
   end
end
```

Use of a hidden dependent property can achieve the desired results.

- In the class definition, set the `OfficeNumber` property attributes to `Dependent` and `Hidden`.
- Create a property set method for `OfficeNumber` that sets the value of the `Location` property.
- Create a property get method for `OfficeNumber` that returns the value of the `Location` property.

While the `OfficeNumber` property is hidden, existing code can continue to access this property. The `Hidden` attribute does not affect access.

Because `OfficeNumber` is dependent, there is no redundancy in storage required by adding the new property. MATLAB does not store or save dependent properties.

Here is the updated class definition.

```
classdef EmployeeList
   properties
      Name
      Email
      Location
   end
   properties (Dependent, Hidden)
      OfficeNumber
   end
   methods
      function obj = set.OfficeNumber(obj,val)
         obj.Location = val;
      end
      function val = get.OfficeNumber(obj)
         val = obj.Location;
      end
```

```
    end
end
```

Saving and Loading EmployeeList Objects

You can load old instances of the `EmployeeList` class in the presence of the new class version. Code that refers to the `OfficeNumber` property continues to work.

Forward and Backward Compatibility

Suppose that you want to be able to load new `EmployeeList` objects into systems that still have the old version of the `EmployeeList` class. To achieve compatibility with old and new versions:

- Define the `OfficeNumber` property as `Hidden`, but not `Dependent`.
- Define the `Location` property as `Dependent`.

In this version of the `EmployeeList` class, the `OfficeNumber` property saves the value used by the `Location` property. Loading an object assigns values of the three original properties (`Name`, `Email`, and `OfficeNumber`), but does not assign a value to the new `Location` property. The lack of the `Location` property in the old class definition is not a problem.

```
classdef EmployeeList
    properties
        Name
        Email
    end
    properties (Dependent)
        Location
    end
    properties (Hidden)
        OfficeNumber
    end
    methods
        function obj = set.Location(obj,val)
            obj.OfficeNumber = val;
        end
        function val = get.Location(obj)
            val = obj.OfficeNumber;
        end
    end
end
```

Update Property When Loading

Suppose that you modify a class so that a property value changes in its form or type. Previously saved objects of the class must be updated when loaded to have a conforming property value.

Consider a class that has an `AccountID` property. Suppose that all account numbers must migrate from eight-digit numeric values to 12-element character arrays.

You can accommodate this change by implementing a `loadobj` method.

The `loadobj` method:

- Tests to determine if the `load` function passed a `struct` or object. All `loadobj` methods must handle both `struct` and object when there is an error in `load`.

- Tests to determine if the `AccountID` number contains eight digits. If so, change it to a 12-element character array by calling the `padAccID` method.

After updating the `AccountID` property, `loadobj` returns a `MyAccount` object that MATLAB loads into the workspace.

```
classdef MyAccount
   properties
      AccountID
   end
   methods
      function obj = padAccID(obj)
         ac = obj.AccountID;
         acstr = num2str(ac);
         if length(acstr) < 12
            obj.AccountID = [acstr,repmat('0',1,12-length(acstr))];
         end
      end
   end
   methods (Static)
      function obj = loadobj(a)
         if isstruct(a)
            obj = MyAccount;
            obj.AccountID = a.AccountID;
            obj = padAccID(obj);
         elseif isa(a,'MyAccount')
            obj = padAccID(a);
         end
      end
   end
end
```

You do not need to implement a `saveobj` method. You are using `loadobj` only to ensure that older saved objects are brought up to date while loading.

Maintaining Compatible Versions of a Class

The `PhoneBookEntry` class uses a combination of techniques to maintain compatibility with new versions of the class.

Suppose that you define a class to represent an entry in a phone book. The `PhoneBookEntry` class defines three properties: `Name`, `Address`, and `PhoneNumber`.

```
classdef PhoneBookEntry
   properties
      Name
      Address
      PhoneNumber
   end
end
```

However, in future releases, the class adds more properties. To provide flexibility, `PhoneBookEntry` saves property data in a `struct` using its `saveobj` method.

```
methods
   function s = saveobj(obj)
      s.Name = obj.Name;
```

```
      s.Address = obj.Address;
      s.PhoneNumber = obj.PhoneNumber;
   end
end
```

The `loadobj` method creates the `PhoneBookEntry` object, which is then loaded into the workspace.

```
methods (Static)
   function obj = loadobj(s)
      if isstruct(s)
         newObj = PhoneBookEntry;
         newObj.Name = s.Name;
         newObj.Address = s.Address;
         newObj.PhoneNumber = s.PhoneNumber;
         obj = newObj;
      else
         obj = s;
      end
   end
end
```

Version 2 of the PhoneBookEntry Class

In version 2 of the `PhoneBookEntry` class, you split the `Address` property into `StreetAddress`, `City`, `State`, and `ZipCode` properties.

With these changes, you could not load a version 2 object in a previous release. However, version 2 employs several techniques to enable compatibility:

- Preserve the `Address` property (which is used in version 1) as a `Dependent` property with private `SetAccess`.
- Define an `Address` property get method (`get.Address`) to build a `char` vector that is compatible with the version 2 `Address` property.
- The `saveobj` method invokes the `get.Address` method to assign the object data to a `struct` that is compatible with previous versions. The `struct` continues to have only an `Address` field built from the data in the new `StreetAddress`, `City`, `State`, and `ZipCode` properties.
- When the `loadobj` method sets the `Address` property, it invokes the property set method (`set.Address`), which extracts the substrings required by the `StreetAddress`, `City`, `State`, and `ZipCode` properties.
- The `Transient` (not saved) property `SaveInOldFormat` enables you to specify whether to save the version 2 object as a `struct` or an object.

```
classdef PhoneBookEntry
   properties
      Name
      StreetAddress
      City
      State
      ZipCode
      PhoneNumber
   end
   properties (Constant)
      Sep = ', '
   end
```

```
    properties (Dependent, SetAccess=private)
       Address
    end
    properties (Transient)
       SaveInOldFormat = false;
    end
    methods (Static)
       function obj = loadobj(s)
          if isstruct(s)
             obj = PhoneBookEntry;
             obj.Name = s.Name;
             obj.Address = s.Address;
             obj.PhoneNumber = s.PhoneNumber;
          else
             obj = s;
          end
       end
    end
    methods
       function address = get.Address(obj)
          address = [obj.StreetAddress,obj.Sep,obj.City,obj.Sep,...
             obj.State,obj.Sep,obj.ZipCode];
       end
       function obj = set.Address(obj,address)
          addressItems = regexp(address,obj.Sep,'split');
          if length(addressItems) == 4
             obj.StreetAddress = addressItems{1};
             obj.City = addressItems{2};
             obj.State = addressItems{3};
             obj.ZipCode = addressItems{4};
          else
             error('PhoneBookEntry:InvalidAddressFormat', ...
                'Invalid address format.');
          end
       end
       function s = saveobj(obj)
          if obj.SaveInOldFormat
             s.Name = obj.Name;
             s.Address = obj.Address;
             s.PhoneNumber = obj.PhoneNumber;
          end
       end
    end
end
```

See Also

Related Examples

- “Modify the Save and Load Process” on page 13-12
- “Object Save and Load”

Initialize Objects When Loading

In this section...

Calling Constructor When Loading Objects

MATLAB does not call the class constructor when loading an object from a MAT-file. However, if you set the `ConstructOnLoad` class attribute to `true`, `load` does call the constructor with no arguments.

Enable `ConstructOnLoad` when you do not want to implement a `loadobj` method, but must perform some actions at construction time. For example, enable `ConstructOnLoad` when you are registering listeners for another object. Ensure that MATLAB can call the class constructor with no arguments without generating an error.

Attributes set on superclasses are not inherited by subclasses. Therefore, MATLAB does not use the value of the superclass `ConstructOnLoad` attribute when loading objects. If you want MATLAB to call the class constructor, set the `ConstructOnLoad` attribute in your specific subclass.

If the constructor requires input arguments, use a `loadobj` method.

Initializing Objects in the loadobj Method

Use a `loadobj` method when the class constructor requires input arguments to perform object initialization.

The `LabResults` class shares the constructor object initialization steps with the `loadobj` method by performing these steps in the `assignStatus` method.

Objects of the `LabResults` class:

- Hold values for the results of tests.
- Assign a status for each value based on a set of criteria.

```
classdef LabResult
   properties
      CurrentValue
   end
   properties (Transient)
      Status
   end
   methods
      function obj = LabResult(cv)
         obj.CurrentValue = cv;
         obj = assignStatus(obj);
      end
      function obj = assignStatus(obj)
         v = obj.CurrentValue;
         if v < 10
            obj.Status = 'Too low';
```

```
            elseif v >= 10 && v < 100
               obj.Status = 'In range';
            else
               obj.Status = 'Too high';
            end
         end
      end
      methods (Static)
         function obj = loadobj(s)
            if isstruct(s)
               cv = s.CurrentValue;
               obj = LabResults(cv);
            else
               obj = assignStatus(s);
            end
         end
      end
end
```

The `LabResults` class uses `loadobj` to determine the status of a given test value. This approach provides a way to:

- Modify the criteria for determining status
- Ensure that objects always use the current criteria

You do not need to implement a `saveobj` method.

See Also

Related Examples

- “Modify the Save and Load Process” on page 13-12
- “Object Save and Load”

Save and Load Objects from Class Hierarchies

In this section...

Saving and Loading Subclass Objects

If the most specific class of an object does not define a `loadobj` or `saveobj` method, this class can inherit `loadobj` or `saveobj` methods from a superclass.

If any class in the hierarchy defines `saveobj` or `loadobj` methods:

- Define `saveobj` for all classes in the hierarchy.
- Call superclass `saveobj` methods from the subclass `saveobj` method because the `save` function calls only the most specific `saveobj` method.
- The subclass `loadobj` method can call the superclass `loadobj`, or other methods as required, to assign values to their properties.

Reconstruct the Subclass Object from a Saved struct

Suppose that you want to save a subclass object by first converting its property data to a `struct` in the class `saveobj` method. Then you reconstruct the object when loaded using its `loadobj` method. This action requires that:

- Superclasses implement `saveobj` methods to save their property data in the `struct`.
- The subclass `saveobj` method calls each superclass `saveobj` method and returns the completed `struct` to the `save` function. Then the `save` function writes the `struct` to the MAT-file.
- The subclass `loadobj` method creates a subclass object and calls superclass methods to assign their property values in the subclass object.
- The subclass `loadobj` method returns the reconstructed object to the `load` function, which loads the object into the workspace.

The following superclass (`MySuper`) and subclass (`MySub`) definitions show how to code these methods.

- The `MySuper` class defines a `loadobj` method to enable an object of this class to be loaded directly.
- The subclass `loadobj` method calls a method named `reload` after it constructs the subclass object.
- `reload` first calls the superclass `reload` method to assign superclass property values and then assigns the subclass property value.

```
classdef MySuper
   properties
      X
      Y
   end
```

```
   methods
      function S = saveobj(obj)
         S.PointX = obj.X;
         S.PointY = obj.Y;
      end
      function obj = reload(obj,S)
         obj.X = S.PointX;
         obj.Y = S.PointY;
      end
   end
   methods (Static)
      function obj = loadobj(S)
         if isstruct(s)
            obj = MySuper;
            obj = reload(obj,S);
         end
      end
   end
end
```

Call the superclass `saveobj` and `loadobj` methods from the subclass `saveobj` and `loadobj` methods.

```
classdef MySub < MySuper
   properties
      Z
   end
   methods
      function S = saveobj(obj)
         S = saveobj@MySuper(obj);
         S.PointZ = obj.Z;
      end
      function obj = reload(obj,S)
         obj = reload@MySuper(obj,S);
         obj.Z = S.PointZ;
      end
   end
   methods (Static)
      function obj = loadobj(S)
         if isstruct(s)
            obj = MySub;
            obj = reload(obj,S);
         end
      end
   end
end
```

See Also

Related Examples

- “Modify the Save and Load Process” on page 13-12
- “Object Save and Load”

Restore Listeners

In this section...

Create Listener with loadobj

Suppose that you create a property listener and want to be able to save and restore the event source and the listener. One approach is to create a listener from the `loadobj` method.

Use Transient Property to Load Listener

The `BankAccount` class stores the account balance and an account status. A `PostSet` listener attached to the `AccountBalance` property controls the account status.

When the `AccountBalance` property value changes, the listener callback determines the account status. Important points include:

- The `BankAccount` class defines the `AccountManagerListener` property to contain the listener handle. This property enables the `loadobj` method to create a listener and return a reference to it in the object that is loaded into the workspace.
- The `AccountManagerListener` property is `Transient` because there is no need to store the listener handle with a `BankAccount` object. Create a listener that is attached to the new `BankAccount` object created during the load process.
- The `AccountBalance` listener sets the `AccountStatus`.
- Only the `AccountManager` class can access `AccountStatus` property.

```
classdef BankAccount < handle
   properties (SetObservable, AbortSet)
      AccountBalance
   end
   properties (Transient)
      AccountManagerListener
   end
   properties (Access = ?AccountManager)
      AccountStatus
   end
   methods
      function obj = BankAccount(initialBalance)
         obj.AccountBalance = initialBalance;
         obj.AccountStatus = 'New Account';
         obj.AccountManagerListener = AccountManager.addAccount(obj);
      end
   end
   methods (Static)
      function obj = loadobj(obj)
         if isstruct(obj) % Handle error
            initialBalance = obj.AccountBalance;
            obj = BankAccount(initialBalance);
```

```
            else
               obj.AccountManagerListener = AccountManager.addAccount(obj);
            end
         end
      end
end
```

Assume the `AccountManager` class provides services for various types of accounts. For the `BankAccount` class, the `AccountManager` class defines two `Static` methods:

- `assignStatus` — Callback for the `AccountBalance` property `PostSet` listener. This method determines the value of the `BankAccount` `AccountStatus` property.
- `addAccount` — Creates the `AccountBalance` property `PostSet` listener. The `BankAccount` constructor and `loadobj` methods call this method.

```
classdef AccountManager
   methods (Static)
      function assignStatus(BA,~)
         if BA.AccountBalance < 0 && BA.AccountBalance >= -100
            BA.AccountStatus = 'overdrawn';
         elseif BA.AccountBalance < -100
            BA.AccountStatus = 'frozen';
         else
            BA.AccountStatus = 'open';
         end
      end
      function lh = addAccount(BA)
         lh = addlistener(BA,'AccountBalance','PostSet', ...
            @(src,evt)AccountManager.assignStatus(BA));
      end
   end
end
```

Using the BankAccount and AccountManager Classes

Create an instance of the `BankAccount` class.

```
ba = BankAccount(100)
```

```
ba =

  BankAccount with properties:

            AccountBalance: 100
    AccountManagerListener: [1x1 event.proplistener]
             AccountStatus: 'New Account'
```

Now set an account value to confirm that the `AccountManager` sets `AccountStatus` appropriately:

```
ba.AccountBalance = -10;
ba.AccountStatus
```

```
ans =

overdrawn
```

See Also

Related Examples

- “Modify the Save and Load Process” on page 13-12
- “Property Attributes” on page 8-8
- “Listen for Changes to Property Values” on page 11-32
- “Object Save and Load”

Enumerations

Named Values

In this section...

Kinds of Predefined Names

MATLAB supports two kinds of predefined names:

- Constant properties
- Enumerations

Constant Properties

Use constant properties when you want a collection of related constant values that can belong to different types (numeric values, character strings, and so on). Define properties with constant values by setting the property `Constant` attribute. Reference constant properties by name whenever you need access to that particular value.

See “Define Class Properties with Constant Values” on page 15-2 for more information.

Enumerations

Use enumerations when you want to create a fixed set of names representing a single type of value. Use this new type in multiple places without redefining it for each class.

You can derive enumeration classes from other classes to inherit the operations of the superclass. For example, if you define an enumeration class that subclasses a MATLAB numeric class like `double` or `int32`, the enumeration class inherits all the mathematical and relational operations that MATLAB defines for those classes.

Using enumerations instead of character strings to represent a value, such as colors (`'red'`), can result in more readable code because:

- You can compare enumeration members with == instead of using `strcmp`
- Enumerations maintain type information, while `char` vectors do not. For example, passing a `char` vector `'red'` to functions means that every function must interpret what `'red'` means. If you define red as an enumeration, the actual value of `'red'` can change (from `[1 0 0]` to `[.93 .14 .14]`, for example) without updating every function that accepts colors, as you would if you defined the color as the `char` vector `'red'`.

Define enumerations by creating an `enumeration` block in the class definition.

See “Define Enumeration Classes” on page 14-4 for more information.

Techniques for Defining Enumerations

Enumerations enable you to define names that represent entities useful to your application, without using numeric values or character strings. All enumerations support equality and inequality operations. Therefore, `switch`, `if`, and several comparison functions like `isequal` and `ismember` work with enumeration members.

You can define enumeration classes in ways that are most useful to your application, as described in the following sections.

Simple Enumerated Names

Simple enumeration classes have no superclasses and no properties. These classes define a set of related names that have no underlying values associated with them. Use this kind of enumeration when you want descriptive names, but your application does not require specific information associated with the name.

See the `WeekDays` class in the “Enumeration Class” on page 14-4 and the “Define Methods in Enumeration Classes” on page 14-5 sections.

Enumerations with Built-In Class Behaviors

Enumeration classes that subclass MATLAB built-in classes inherit most of the behaviors of those classes. For example, an enumeration class derived from the `double` class inherits the mathematical, relational, and set operations that work with variables of the class.

Enumerations do not support the colon (`:`) operator, even if the superclass does.

Enumerations with Properties for Member Data

Enumeration classes that do not subclass MATLAB built-in numeric and logical classes can define properties. These classes can define constructors that set each member's unique property values.

The constructor can save input arguments in property values. For example, a `Color` class can specify a `Red` enumeration member color with three (Red, Green, Blue) values:

```
enumeration
   Red (1,0,0)
end
```

See Also

Related Examples

- “Enumeration Class Restrictions” on page 14-26
- “Enumerations Derived from Built-In Classes” on page 14-27
- “Enumerations That Encapsulate Data” on page 14-37

Define Enumeration Classes

In this section...

Enumeration Class

Create an enumeration class by adding an `enumeration` block to a class definition. For example, the `WeekDays` class enumerates a set of days of the week.

```
classdef WeekDays
   enumeration
      Monday, Tuesday, Wednesday, Thursday, Friday
   end
end
```

To execute the MATLAB code in the following sections, place the `WeekDays` class definition in a `.m` file on your path.

Construct an Enumeration Member

Refer to an enumeration member using the class name and the member name:

`ClassName.MemberName`

For example, assign the enumeration member `WeekDays.Tuesday` to the variable `today`:

```
today = WeekDays.Tuesday;
```

`today` is a variable of class `WeekDays`:

```
whos
```

```
  Name       Size            Bytes  Class       Attributes

  today      1x1               104  WeekDays
```

```
today
```

```
today =

    Tuesday
```

Convert to Superclass Value

If an enumeration class specifies a superclass, you can convert an enumeration object to the superclass by passing the object to the superclass constructor. However, the superclass constructor

must be able to accept its own class as input and return an instance of the superclass. MATLAB built-in numeric classes, such as `uint32`, allow this conversion.

For example, the `Bearing` class derives from the `uint32` built-in class:

```
classdef Bearing < uint32
   enumeration
      North (0)
      East  (90)
      South (180)
      West  (270)
   end
end
```

Assign the `Bearing.East` member to the variable `a`:

```
a = Bearing.East;
```

Pass `a` to the superclass constructor and return a `uint32` value:

```
b = uint32(a);
whos
```

```
  Name      Size             Bytes  Class      Attributes

  a         1x1                  4  Bearing
  b         1x1                  4  uint32
```

The `uint32` constructor accepts an object of the subclass `Bearing` and returns an object of class `uint32`.

Define Methods in Enumeration Classes

Define methods in an enumeration class like any MATLAB class. For example, define a method called `isMeetingDay` for the `WeekDays` enumeration class. The use case is that the user has a recurring meeting on Tuesdays. The method checks if the input argument is an instance of the `WeekDays` member `Tuesday`.

```
classdef WeekDays
   enumeration
      Monday, Tuesday, Wednesday, Thursday, Friday
   end
   methods
      function tf = isMeetingDay(obj)
         tf = WeekDays.Tuesday == obj;
      end
   end
end
```

Call `isMeetingDay` with an instance of the `WeekDays` class:

```
today = WeekDays.Tuesday;
today.isMeetingDay
```

```
ans =

     1
```

You can also use the enumeration member as a direct input to the method:

```
isMeetingDay(WeekDays.Wednesday)
```

```
ans =

     0
```

Define Properties in Enumeration Classes

Add properties to an enumeration class when you must store data related to the enumeration members. Set the property values in the class constructor. For example, the `SyntaxColors` class defines three properties. The class constructor assigns the values of the input arguments to the corresponding properties when you reference a class member.

```
classdef SyntaxColors
   properties
      R
      G
      B
   end
   methods
      function c = SyntaxColors(r, g, b)
         c.R = r; c.G = g; c.B = b;
      end
   end
   enumeration
      Error   (1, 0, 0)
      Comment (0, 1, 0)
      Keyword (0, 0, 1)
      String  (1, 0, 1)
   end
end
```

When you refer to an enumeration member, the constructor initializes the property values:

```
e = SyntaxColors.Error;
e.R
```

```
ans =

     1
```

Because `SyntaxColors` is a value class (it does not derive from `handle`), only the class constructor can set property values:

```
e.R = 0
```

```
You cannot set the read-only property 'R' of SyntaxColors.
```

For more information on enumeration classes that define properties, see “Mutable Handle vs. Immutable Value Enumeration Members” on page 14-32.

Enumeration Class Constructor Calling Sequence

Each statement in an enumeration block is the name of an enumeration member, optionally followed by an argument list. If the enumeration class defines a constructor, MATLAB calls the constructor to create the enumerated instances.

MATLAB provides a default constructor for all enumeration classes that do not explicitly define a constructor. The default constructor creates an instance of the enumeration class:

- Using no input arguments, if the enumeration member defines no input arguments
- Using the input arguments defined in the enumeration class for that member

For example, the input arguments for the `Bool` class are `0` for `Bool.No` and `1` for `Bool.Yes`.

```
classdef Bool < logical
   enumeration
      No  (0)
      Yes (1)
   end
end
```

The values of `0` and `1` are of class `logical` because the default constructor passes the argument to the first superclass. That is, this statement:

```
n = Bool.No;
```

Results in a call to `logical` that is equivalent to the following statement in a constructor:

```
function obj = Bool(val)
   obj@logical(val)
end
```

MATLAB passes the member argument only to the first superclass. For example, suppose `Bool` derived from another class:

```
classdef Bool < logical & MyBool
   enumeration
      No  (0)
      Yes (1)
   end
end
```

The `MyBool` class can add some specialized behavior:

```
classdef MyBool
   methods
      function boolValues = testBools(obj)
         ...
      end
   end
end
```

The default `Bool` constructor behaves as if defined like this function:

- Argument passed to first superclass constructor
- No arguments passed to subsequent constructors

```
function obj = Bool(val)
   obj@logical(val)
   obj@MyBool
end
```

See Also

Related Examples

- “Refer to Enumerations” on page 14-9
- “Operations on Enumerations” on page 14-16

Refer to Enumerations

In this section...

Instances of Enumeration Classes

Enumeration members are instances of the enumeration class. You can assign enumeration members to variables and form arrays of enumeration members. If an enumeration class derives from a superclass, you can substitute an enumeration member for an instance of the superclass.

The `WeekDays` class defines enumeration members for five days of the week.

```
classdef WeekDays
   enumeration
      Monday, Tuesday, Wednesday, Thursday, Friday
   end
end
```

Create objects of the `WeekDays` class representing specific days.

```
today = WeekDays.Monday;
tomorrow = WeekDays.Tuesday;
```

The variables `today` and `tomorrow` are objects of the `WeekDays` class.

The `PPM` class defines three enumeration members. Each member has an associated numeric value derived from the class superclass.

```
classdef PPM < double
   enumeration
      High (1000)
      Medium (100)
      Low (10)
   end
end
```

Assign an enumeration member to a variable.

```
level = PPM.High;
```

When you substitute enumeration members for instances of the superclass, MATLAB coerces the enumeration member to the superclass. For example, add a numeric value to an enumeration member of the `PPM` class.

```
levelNew = level + 100
```

```
levelNew =

        1100
```

The result is of class `double`.

```
whos
```

```
  Name          Size            Bytes  Class    Attributes

  level         1x1               108  PPM
  levelNew      1x1                 8  double
```

You can substitute superclass values for corresponding enumeration members. For example, pass one of the numeric values defined in the enumeration class to the `PPMSwitch` function.

```
function PPMSwitch(ppm)
   switch ppm
      case PPM.Low
         disp Low
      case PPM.Medium
         disp Medium
      case PPM.High
         disp High
   end
end

PPMSwitch(100)
```

```
Medium
```

You can also use an enumeration member directly:

```
PPMSwitch(PPM.Medium)
```

```
Medium
```

For information on operations you can perform on enumeration class instances, see “Operations on Enumerations” on page 14-16.

Conversion of Characters to Enumerations

Enumeration classes can convert `char` vectors to enumeration members when the `char` vector represents an enumeration member defined by the class. This conversion enables you to pass a valid `char` vector or a cell array of `char` vectors when enumerations are expected.

Use a `char` vector instead of a direct reference to an enumeration member when you want to use a simple character string to specify an enumeration member. However, specifying an enumeration member directly eliminates the conversion from `char` to enumeration.

Enumeration classes provide a converter function using the constructor syntax.

```
today = WeekDays('Tuesday');
```

Because the `char` vector `'Tuesday'` matches the enumeration member `WeekDays.Tuesday`, the `Weekdays` `char` method can perform the conversion.

```
class(today)
```

```
ans =

WeekDays
```

Create an enumeration array using the `WeekDay` class constructor and a cell array of `char` vectors.

```
wd = WeekDays({'Monday','Wednesday','Friday'})
```

```
wd =

    Monday       Wednesday    Friday
```

```
class(wd)
```

```
ans =

WeekDays
```

All `char` vectors in the cell array must correspond to an enumeration member defined by the class.

Coercion of char to Enumerations

MATLAB coerces `char` vectors into enumeration members when the dominant argument is an enumeration. Because user-defined classes are dominant over the `char` class, MATLAB attempts to convert the `char` vector to a member of the enumeration class.

Create an enumeration array. Then insert a `char` vector that represents an enumeration member into the array.

```
a = [WeekDays.Monday,WeekDays.Wednesday,WeekDays.Friday]
```

```
a =

    Monday       Wednesday    Friday
```

Add a `char` vector to the `WeekDays` array.

```
a(end+1) = 'Tuesday'
```

```
a =

    Monday       Wednesday    Friday       Tuesday
```

MATLAB coerces the `char` vector to a `WeekDays` enumeration member.

```
class(a)
```

```
ans =

WeekDays
```

Substitute Enumeration Members for char Vectors

You can use enumeration members in place of `char` vectors in cases where functions require `char` vectors. For example, this call to `sprintf` expects a `char` vector, designated by the `%s` format specifier.

```
sprintf('Today is %s',WeekDays.Friday)
```

```
ans =

Today is Friday
```

The automatic conversion of enumeration classes to `char` enable you to use enumeration members in this case.

Enumeration Arrays

Create enumeration arrays by:

- Concatenating enumeration members using `[]`
- Assigning enumeration members to an array using indexed assignment

Create an enumeration array of class `WeekDays` by concatenating enumeration members:

```
wd = [WeekDays.Tuesday,WeekDays.Wednesday,WeekDays.Friday];
```

Create an enumeration array of class `WeekDays` by indexed assignment:

```
a(1) = WeekDays.Tuesday;
a(2) = WeekDays.Wednesday;
a(3) = WeekDays.Friday;
```

Mixed Enumeration Members and char Vectors

You can concatenate enumeration members and `char` vectors as long as the `char` vector represents an enumeration member.

```
clear a
a = [WeekDays.Wednesday,'Friday'];
class(a)

ans =

WeekDays
```

You can also assign a `char` vector to an enumeration array:

```
clear a
a(1) = WeekDays.Wednesday;
a(2) = 'Friday';
class(a)

ans =

WeekDays
```

Default Enumeration Member

The default member of an enumeration class is the first enumeration member defined in the enumeration block. For the `WeekDays` class, the default enumeration member is `WeekDays.Monday`.

```
classdef WeekDays
   enumeration
      Monday, Tuesday, Wednesday, Thursday, Friday
   end
end
```

MATLAB allows assignment to any element of an array, even if the array variable does not previously exist. To fill in unassigned array elements, MATLAB uses the default enumeration member.

For example, assign a value to element 5 of an array, `a`:

```
clear a
a(5) = WeekDays.Tuesday;
```

MATLAB must initialize the values of array elements `a(1:4)` with the default enumeration member. The result of the assignment to the fifth element of the array `a` is:

```
a
```

```
a =

    Monday        Monday        Monday        Monday        Tuesday
```

See Also

Related Examples

- “Operations on Enumerations” on page 14-16

Enumerations for Property Values

In this section...

Syntax for Property/Enumeration Definition

You can restrict the values that are allowed for a property to members of an enumeration class. Define the property as restricted to a specific enumeration class in the class definition using this syntax:

```
properties
   PropName EnumerationClass
end
```

This syntax restricts values of *`PropName`* to members of the enumeration class *`EnumerationClass`*.

Example of Restricted Property

For example, the `Days` class defines a property named `Today`. The allowed values for the `Today` property are enumeration members of the `WeekDays` class.

The `WeekDays` class defines the enumerations:

```
classdef WeekDays
   enumeration
      Monday, Tuesday, Wednesday, Thursday, Friday
   end
end
```

Use the `WeekDays` enumerations to restrict the allowed values of the `Today` property:

```
classdef Days
   properties
      Today WeekDays
   end
end
```

Create an object of the `Days` class.

```
d = Days;
d.Today = WeekDays.Tuesday;
```

```
d =

  Days with properties:

    Today: Tuesday
```

Representing Enumeration Members with char Vectors

The automatic conversion feature enables users of the `Days` class to assign values to the `Today` property as either enumeration members, `char` vectors, or string scalars. The `Today` property is

restricted to members of the `WeekDays` enumeration class. Therefore, you can assign a `char` vector that represents a member of the `WeekDays` class.

```
d = Days;
d.Today = 'Tuesday';
```

Also, you can use a string scalar:

```
d = Days;
d.Today = "Tuesday";
```

For more information on restricting property values, see “Validate Property Values” on page 8-18 and “Property Class and Size Validation” on page 8-23.

Operations on Enumerations

In this section...

Operations Supported by Enumerations

You can use logical, set membership, and string comparison operations on enumerations. These operations also support the use of enumerations in conditional statements, such as `switch` and `if` statements. The `string` and `char` functions enable you to convert enumeration members to strings and `char` vectors.

Example Enumeration Class

This topic uses the `WeekDays` class to illustrate how to perform operations on enumerations. The `WeekDays` class defines members that enumerate days of the week.

```
classdef WeekDays
   enumeration
      Monday, Tuesday, Wednesday, Thursday, Friday
   end
end
```

For information on defining enumerations, see “Define Enumeration Classes” on page 14-4.

Default Methods

Enumeration classes have the following default methods:

```
methods('WeekDays')
```

```
Methods for class WeekDays:

WeekDays   char       intersect  ne         setxor     strcmpi    strncmp    union
cellstr    eq         ismember   setdiff    strcmp     string     strncmpi
```

The `WeekDays` method converts text formats into enumerations. Supported formats include strings, `char` vectors, string arrays, and cell arrays of `char` vectors. For example:

```
f = WeekDays(["Monday" "Friday"])
```

```
f =

  1×2 WeekDays enumeration array

    Monday    Friday
```

Convert Enumeration Members to Strings or char Vectors

Conversion of enumeration members to strings and `char` vectors is useful for creating text that contains enumeration member names. For example, use the `string` function to convert an enumeration member to a string and include it in a sentence:

```
string(WeekDays.Monday) + " is our meeting day."
```

```
ans =

    "Monday is our meeting day."
```

Use the `char` function in a similar way:

```
['Today is ' char(WeekDays.Friday) '.']
```

```
ans =

    'Today is Friday.'
```

Convert Enumeration Arrays to String Arrays or Cell Arrays of char Vectors

Use the `string` function to convert an enumeration array into a string array:

```
sa = [WeekDays.Tuesday WeekDays.Thursday];
string(sa)
```

```
ans =

  1×2 string array

    "Tuesday"    "Thursday"
```

Use `cellstr` to convert an enumeration array to a cell array of `char` vectors.

```
ca = cellstr([WeekDays.Tuesday WeekDays.Thursday]);
class(ca)
```

```
ans =

    'cell'
```

Both cells in the cell array contain `char` vectors:

```
class([ca{1:2}])
```

```
ans =

    'char'
```

Relational Operations with Enumerations, Strings, and char Vectors

You can compare enumeration instances with `char` vectors and strings using the relational operators `eq` (==) and `ne` (~=), as well as the `isequal` method.

Relational Operators eq and ne

Use `eq` and `ne` to compare enumeration members with text values. For example, you can compare an enumeration member with a string:

```
today = WeekDays.Friday;
today == "Friday"
```

```
ans =

  logical

   1
```

Compare an enumeration array to one `char` vector. The return value is a logical array indicating which members of the enumeration array are equivalent to the `char` vector:

```
wd = [WeekDays.Monday WeekDays.Wednesday WeekDays.Friday];
wd == 'Friday'
```

```
ans =

  1×3 logical array

   0   0   1
```

This example uses the `ne` function to compare the corresponding elements of an enumeration array and a string array of equal length:

```
sa = ["Monday" "Wednesday" "Friday"];
md = [WeekDays.Tuesday WeekDays.Thursday WeekDays.Friday];
md ~= sa
```

```
ans =

  1×3 logical array

   1   1   0
```

The `char` vector `Wednesday` is equal to (==) the enumeration member `WeekDays.Wednesday`. You can use this equality in conditional statements:

```
today = 'Wednesday';
   ...
if today == WeekDays.Wednesday
   disp('Team meeting at 2:00')
end
```

isequal Method

The `isequal` method also enables comparisons between enumeration instances and strings, character vectors, string arrays, and cell arrays of character vectors.

```
a = WeekDays.Monday;
isequal(a,"Monday")

ans =

     logical

     1
```

When comparing an enumeration array to a single item, the behavior of `isequal` differs slightly from `eq` and `ne`. The `isequal` method requires that the two values being compared are the same size. Therefore, `isequal` returns false when comparing an enumeration array to a `char` vector or string scalar, even if the text matches one of the enumeration members in the array.

```
wd = [WeekDays.Monday WeekDays.Wednesday WeekDays.Friday];
isequal(wd,"Friday")

ans =

     logical

     0
```

Enumerations in switch Statements

Equality (`eq`) and inequality (`ne`) functions enable you to use enumeration members in `switch` statements. For example, using the `WeekDays` enumeration defined previously, construct a switch statement:

```
function c = Reminder(day)
   % Add error checking here
   switch(day)
      case WeekDays.Monday
         c = 'No meetings';
      case WeekDays.Tuesday
         c = 'Department meeting at 10:00';
      case {WeekDays.Wednesday WeekDays.Friday}
         c = 'Team meeting at 2:00';
      case WeekDays.Thursday
         c = 'Volleyball night';
   end
end
```

Pass a member of the `WeekDays` enumeration class to the `Reminder` function:

```
today = WeekDays.Wednesday;
Reminder(today)

ans =

Team meeting at 2:00
```

For more information, see “Objects in Conditional Statements” on page 5-20.

Substitute Strings or char Vectors

You can use strings or `char` vectors to represent specific enumeration members:

```
function c = Reminder2(day)
   switch(day)
      case 'Monday'
         c = 'Department meeting at 10:00';
      case 'Tuesday'
         c = 'Meeting Free Day!';
      case {'Wednesday' 'Friday'}
         c = 'Team meeting at 2:00';
      case 'Thursday'
         c = 'Volleyball night';
   end
end
```

Although you can use `char` vectors or strings instead of specifying enumerations explicitly, MATLAB must convert the text format to an enumeration. Eliminate the need for this conversion if it is not necessary.

Enumeration Set Membership

Enumeration classes provide methods to determine set membership.

- `ismember` — True for elements of an enumeration array if in a set
- `setdiff` — Set difference for enumeration arrays
- `intersect` — Set intersection for enumeration arrays
- `setxor` — Set exclusive-or for enumeration arrays
- `union` — Set union for enumeration arrays

Determine if today is a meeting day for your team. Create a set of enumeration members corresponding to the days on which the team has meetings.

```
today = WeekDays.Tuesday;
teamMeetings = [WeekDays.Wednesday WeekDays.Friday];
```

Use `ismember` to determine if `today` is part of the `teamMeetings` set:

```
ismember(today,teamMeetings)
```

```
ans =
     0
```

Mixed Sets of Enumeration and Text

If you pass both enumeration members and text values to an enumeration class method, the class attempts to convert the text value to the class of the enumeration.

Determine if the `char` vector `'Friday'` is a member of the enumeration array.

```
teamMeetings = [WeekDays.Wednesday WeekDays.Friday];
ismember('Friday',teamMeetings)
```

```
ans =

  logical

   1
```

Determine if the enumeration member is a member of the string array.

```
ismember(WeekDays.Friday,["Wednesday" "Friday"])
```

```
ans =

  logical

   1
```

Enumeration Text Comparison Methods

Enumeration classes provide methods to compare enumeration members with text. One of the arguments to the string comparison methods must be a `char` vector or a string. Comparing two enumeration members returns `false`.

- `strcmp` — Compare enumeration members
- `strncmp` — Compare first n characters of enumeration members
- `strcmpi` — Case insensitive comparison of enumeration members
- `strncmpi` — Case insensitive first n character comparison of enumeration members

Comparing Enumeration Member with Strings or char Vectors

The string comparison methods can compare enumeration members with `char` vectors and strings.

```
today = WeekDays.Tuesday;
strcmp(today,'Friday')
```

```
ans =

     0
```

```
strcmp(today,"Tuesday")
```

```
ans =

     1
```

Get Information About Enumerations

Obtain information about enumeration classes using the `enumeration` function. For example:

```
enumeration WeekDays
```

```
Enumeration members for class 'WeekDays':

    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
```

For more information on how class introspection works with enumerations, see “Metaclass EnumeratedValues Property” on page 16-7.

Testing for an Enumeration

To determine if a value is an enumeration, use the `isenum` function. For example:

```
today = WeekDays.Wednesday;
isenum(today)
```

```
ans =

     1
```

`isenum` returns `true` for empty enumeration objects:

```
noday = WeekDays.empty;
isenum(noday)
```

```
ans =

     1
```

To determine if a class is an enumeration class, use the `meta.class` object.

```
today = WeekDays.Wednesday;
mc = metaclass(today);
mc.Enumeration
```

```
ans =

     1
```

See Also

Related Examples

- “Enumeration Class Restrictions” on page 14-26

Hide Enumeration Members

Hiding enumeration members enables class authors to change enumeration member names without causing incompatibilities in existing code. To hide members, create an enumeration block that sets the `Hidden` attribute. Members defined in a `Hidden` enumeration block are not visible when enumeration members are queried using the `enumeration` function.

When an enumeration class derives from another class, such as a numeric or logical class, then each member can have a value associated with it. If two members have the same value assigned to them, then the member defined first in the class definition masks the second member. Both names are valid enumeration members, but the first one defined is the primary member. While masking makes it possible to use one member name in place of another, it does not hide the secondary name from the class users.

Using the `Hidden` attribute removes the masked member names from user view. For example, the `HighlightColor` class defines enumeration members that represent syntax highlighting colors.

```
classdef HighlightColor < int32
    enumeration
        red   (1)
        green (2)
        blue  (3)
    end
end
```

A new version of the class uses more descriptive member names, but the class needs to avoid breaking existing code that uses the original member names, red, green, and blue. Using the `Hidden` attribute for enumeration members enables the class to hide the original members.

```
classdef HighlightColor < int32
    enumeration
        error   (1)
        comment (2)
        keyword (3)
    end
    enumeration (Hidden)
        red   (1)
        green (2)
        blue  (3)
    end
end
```

Code that uses the original member names continues to work. For example, existing references to the now-hidden member `HighlightColor.blue` is compatible with the same-valued nonhidden member `HighlightColor.keyword`.

```
a = HighlightColor.blue
```

```
a =

  HighlightColor enumeration

    keyword
```

```
a == HighlightColor.Keyword
```

```
ans =
```

```
  logical

   1
```

For enumeration members that represent values, the first member defined in the class is the primary member for that value. For example, in the `HighlightColor` class, `keyword` is the primary member and `blue` is the secondary member, both representing the value 3. Typically, the primary member is not hidden while the secondary member is hidden. However, if the class design requires that the primary member is hidden, then the secondary member must be hidden too.

Hide Pure Enumerations

Pure enumeration members have no underlying values, so there is no way to identify one member as a replacement for another. However, you can use the `Hidden` attribute to remove a member from the user view while avoiding incompatibilities with existing uses of the hidden member.

For example, the `PCComponents` class defines enumerations that are used in an online form for a computer order. While the `FloppyDrive` component is obsolete, the enumeration member can remain in the class as a hidden member. The form can exclude `FloppyDrive` from the list of choices, but the class author can keep this member available so that existing forms that refer to `FloppyDrive` are still valid.

```
classdef PCComponents
    enumeration
        USBSlots
        CDPlayer
    end
    enumeration (Hidden)
        FloppyDrive
    end
end
```

Find Hidden Enumeration Members

Find information about hidden enumeration members using class metadata. The `meta.EnumeratedValue` class provides information on enumeration members. For example, accessing the metadata for the `HighlightColor` class used in preceding examples can show the names of hidden members.

```
mc =?HighlightColor
```

```
mc =

  class with properties:

                    Name: 'HighlightColor'
             Description: ''
      DetailedDescription: ''
                  Hidden: 0
                  Sealed: 0
                Abstract: 0
             Enumeration: 1
          ConstructOnLoad: 0
         HandleCompatible: 0
          InferiorClasses: {[1×1 meta.class]}
```

```
        ContainingPackage: [0×0 meta.package]
                  Aliases: [0×1 string]
     RestrictsSubclassing: 0
             PropertyList: [0×1 meta.property]
               MethodList: [140×1 meta.method]
                EventList: [0×1 meta.event]
    EnumerationMemberList: [6×1 meta.EnumeratedValue]
           SuperclassList: [1×1 meta.class]
```

Each enumeration member is describe by a `meta.EnumeratedValue` object that is contained in the `EnumerationMemberList` property. For example, the fourth element in the `EnumerationMemberList` array contains the `meta.EnumerationValue` object for the member with the name `red`.

```
mc.EnumerationMemberList(4)
```

```
ans =

  EnumeratedValue with properties:

                  Name: 'red'
           Description: ''
    DetailedDescription: ''
                Hidden: 1
```

To list the names of all hidden members, use the handle class `findobj` method:

```
 findobj(mc.EnumerationMemberList,'Hidden',true).Name
```

```
ans =

    'red'


ans =

    'green'


ans =

    'blue'
```

See Also

`findobj` | `enumeration`

Enumeration Class Restrictions

Enumeration classes restrict certain aspects of their use and definition:

- Enumeration classes are implicitly `Sealed`. You cannot define a subclass of an enumeration class because doing so would expand the set.
- The properties of value-based enumeration classes are immutable. Only the constructor can assign property values. MATLAB implicitly defines the `SetAccess` attributes of all properties defined by value-based enumeration classes as `immutable`. You cannot set the `SetAccess` attribute to any other value.
- All properties inherited by a value-based enumeration class that are not defined as `Constant` must have `immutable` `SetAccess`.
- The properties of handle-based enumeration classes are mutable. You can set property values on instances of the enumeration class. See “Mutable Handle vs. Immutable Value Enumeration Members” on page 14-32.
- An enumeration member cannot have the same name as a property, method, or event defined by the same class or inherited from a superclass. For example, an enumeration class that inherits from a built-in numeric superclass inherits the `full` method from the superclass, so "full" is not a valid member name.
- Enumerations do not support colon (`a:b`) operations. For example, `FlowRate.Low:FlowRate.High` causes an error even if the `FlowRate` class derives from a numeric superclass.
- Classes that define enumerations cannot restrict properties of the same class to an enumeration type. Create a separate enumeration class to restrict property values to an enumeration. For information on restricting property values, see “Example of Restricted Property” on page 14-14.
- If the primary enumeration member sets the `Hidden` attribute, then the secondary member (one with the same underlying value) must also set the `Hidden` attribute. For more information, see “Hide Enumeration Members” on page 14-23.

See Also

Related Examples

- “Enumerations Derived from Built-In Classes” on page 14-27

Enumerations Derived from Built-In Classes

In this section...

Subclassing Built-In Classes

Enumeration classes can subclass MATLAB built-in classes. Deriving an enumeration class from built-in classes is useful to extend the usefulness of the enumeration members.

- Enumerations inherit functionality from the built-in class.
- You can associate a numeric or logical value with enumeration members.

For a more basic discussion of enumeration classes, see “Define Enumeration Classes” on page 14-4.

Derive Enumeration Class from Numeric Class

Note Enumeration classes derived from built-in numeric and logical classes cannot define properties.

If an enumeration class subclasses a built-in numeric class, the subclass inherits ordering and arithmetic operations that you can apply to the enumerated names.

For example, the `Results` class subclasses the `int32` built-in class. This class associates an integer value with each of the four enumeration members — `First`, `Second`, `Third`, and `NoPoints`.

```
classdef Results < int32
   enumeration
      First   (100)
      Second  (50)
      Third   (10)
      NoPlace (0)
   end
end
```

The enumeration member inherits the methods of the `int32` class (except the colon operator). Use these enumerations like numeric values (summed, sorted, averaged).

```
isa(Results.Second,'int32')
```

```
ans =

     1
```

For example, use enumeration names instead of numbers to rank two teams:

```
Team1 = [Results.First, Results.NoPlace, Results.Third, Results.Second];
Team2 = [Results.Second, Results.Third, Results.First, Results.First];
```

Perform `int32` operations on these `Results` enumerations:

```
sum(Team1)

ans =

   160

mean(Team1)

ans =

    40

sort(Team2,'descend')

ans =

    First      First      Second     Third

Team1 > Team2

ans =

     1     0     0     0

sum(Team1) < sum(Team2)

ans =

     1
```

How to Create Enumeration Instances

When you first refer to an enumeration class that derives from a built-in class such as, `int32`, MATLAB passes the input arguments associated with the enumeration members to the superclass constructor. For example, referencing the `Second` `Results` member, defined as:

```
Second (50)
```

means that MATLAB calls:

```
int32(50)
```

to initialize the `int32` aspect of this `Results` object.

How to Alias Enumeration Names

Enumeration classes that derive from MATLAB built-in numeric and logical classes can define more than one name for an underlying value. The first name in the enumeration block with a given underlying value is the actual name for that underlying value and subsequent names are aliases.

Specify aliased names with the same superclass constructor argument as the actual name:

```
classdef Bool < logical
   enumeration
      No  (0)
      Yes (1)
      off (0)
```

```
      on  (1)
   end
end
```

For example, the actual name of an instance of the `Bool.off` enumeration member is `No`:

```
a = Bool.No
```

```
a =

    No
```

```
b = Bool.off
```

```
b =

    No
```

Superclass Constructor Returns Underlying Value

The actual underlying value associated with an enumeration member is the value returned by the built-in superclass. For example, consider the `Bool` class defined with constructor arguments that are of class `double`:

```
classdef Bool < logical
   enumeration
      No  (0)
      Yes (100)
   end
end
```

This class derives from the built-in `logical` class. Therefore, underlying values for an enumeration member depend only on what value `logical` returns when passed that value:

```
a = Bool.Yes
```

```
a =

    Yes
```

```
logical(a)
```

```
ans =

     1
```

How to Subclass Numeric Built-In Classes

The `FlowRate` enumeration class defines three members, `Low`, `Medium`, and `High`.

```
classdef FlowRate < int32
   enumeration
      Low    (10)
      Medium (50)
      High   (100)
   end
end
```

Reference an instance of an enumeration member:

```
setFlow = FlowRate.Medium;
```

This statement causes MATLAB to call the default constructor with the argument value of `50`. MATLAB passes this argument to the first superclass constructor (`int32(50)` in this case). The result is an underlying value of `50` as a 32-bit integer for the `FlowRate.Medium` member.

Because `FlowRate` subclasses a built-in numeric class (`int32`), this class cannot define properties. However `FlowRate` inherits `int32` methods including a converter method. Programs can use the converter to obtain the underlying value:

```
setFlow = FlowRate.Medium;
int32(setFlow)
```

```
ans =

          50
```

Default Converter

If an enumeration is a subclass of a built-in numeric class, you can convert from built-in numeric data to the enumeration using the name of the enumeration class. For example:

```
a = Bool(1)
```

```
a =

    Yes
```

An enumerated class also accepts enumeration members of its own class as input arguments:

```
Bool(a)
```

```
ans =

    Yes
```

The converter returns an object of the same size as in input:

```
Bool([0,1])
```

```
ans =

    No     Yes
```

Create an empty enumeration array using the `empty` static method:

```
Bool.empty
```

```
ans =

  0x0 empty Boolean enumeration.
```

See Also

Related Examples

- “Mutable Handle vs. Immutable Value Enumeration Members” on page 14-32
- “Fundamental MATLAB Classes”

Mutable Handle vs. Immutable Value Enumeration Members

In this section...

Select Handle- or Value-Based Enumerations

Use a handle enumeration to enumerate a set of objects whose state can change over time. Use a value enumeration to enumerate a set of abstract (and immutable) values. For information about handle and value classes, see “Comparison of Handle and Value Classes” on page 7-2.

Value-Based Enumeration Classes

A value-based enumeration class has a fixed set of specific values. Modify these values by changing the values of properties. Doing so expands or changes the fixed set of values for this enumeration class.

Inherited Property SetAccess Must Be Immutable

Value-based enumeration classes implicitly define the `SetAccess` attributes of all properties as `immutable`. You cannot set the `SetAccess` attribute to any other value.

However, all superclass properties must explicitly define property `SetAccess` as `immutable`.

Enumeration Members Remain Constant

An instance of a value-based enumeration class is unique until the class is cleared and reloaded. For example, given this class:

```
classdef WeekDays
   enumeration
      Monday, Tuesday, Wednesday, Thursday, Friday
   end
end
```

MATLAB considers `a` and `b` as equivalent:

```
a = WeekDays.Monday;
b = WeekDays.Monday;
isequal(a,b)
```

```
ans =

     1
```

```
a == b
```

```
ans =

     1
```

Enumeration Member Properties Remain Constant

Value-based enumeration classes that define properties are immutable. For example, the `Colors` enumeration class associates RGB values with color names.

```
classdef Colors
   properties
      R = 0
      G = 0
      B = 0
   end
   methods
      function c = Colors(r,g,b)
         c.R = r; c.G = g; c.B = b;
      end
   end
   enumeration
      Red   (1, 0, 0)
      Green (0, 1, 0)
      Blue  (0, 0, 1)
   end
end
```

The constructor assigns the input arguments to `R`, `G`, and `B` properties:

```
red = Colors.Red;
[red.R,red.G,red.B]
```

```
ans =

     1     0     0
```

You cannot change a property value:

```
red.G = 1;
```

```
You cannot set the read-only property 'G' of Colors.
```

Handle-Based Enumeration Classes

Handle-based enumeration classes that define properties are mutable. Derive enumeration classes from the `handle` class when you must be able to change property values on instances of that class.

Note You cannot derive an enumeration class from `matlab.mixin.Copyable` because the number of instances you can create are limited to the ones defined inside the enumeration block.

An Enumeration Member Remains Constant

Given a handle-based enumeration class with properties, changing the property value of an instance causes all references to that instance to reflect the changed value.

For example, the `HandleColors` enumeration class associates RGB values with color names, the same as the `Colors` class in the previous example. However, `HandleColors` derives from `handle`:

```
classdef HandleColors < handle
   properties
      R = 0
      G = 0
      B = 0
   end
   methods
      function c = HandleColors(r, g, b)
         c.R = r; c.G = g; c.B = b;
      end
   end
   enumeration
      Red   (1, 0, 0)
      Green (0, 1, 0)
      Blue  (0, 0, 1)
   end
end
```

Create an instance of `HandleColors.Red` and return the value of the `R` property:

```
a = HandleColors.Red;
a.R
```

```
ans =

    1
```

MATLAB constructs the `HandleColors.Red` enumeration member, which sets the `R` property to `1`, the `G` property to `0`, and the `B` property to `0`.

Change the value of the `R` property to `0.8`:

```
a.R = 0.8;
```

After setting the value of the `R` property to `0.8`, create another instance, `b`, of `HandleColors.Red`:

```
b = HandleColors.Red;
b.R
```

```
ans =

    0.8000
```

The value of the `R` property of the newly created instance is also `0.8`. A MATLAB session has only one value for any enumeration member at any given time.

Clearing the workspace variables does not change the current definition of the enumeration member `HandleColors.Red`:

```
clear
a = HandleColors.Red;
a.R
```

```
ans =

    0.8000
```

Clear the class to reload the definition of the `HandleColors` class:

```
clear classes
a = HandleColors.Red;
a.R
```

```
ans =

     1
```

To prevent reassignment of a given property value, set that property's `SetAccess` attribute to `immutable`.

Equality of Handle-Based Enumerations

Assign two variables to a particular enumeration member:

```
a = HandleColors.Red;
b = HandleColors.Red;
```

Compare `a` and `b` using `isequal`:

```
isequal(a,b)
```

```
ans =

     1
```

The property values of `a` and `b` are the same, so `isequal` returns `true`. However, unlike handle classes that are not enumeration classes, `a` and `b` are the same handle because there is only one enumeration member. Determine handle equality using `==` (the handle `eq` method).

```
a == b
```

```
ans =

     1
```

See the `handle` `eq` method for information on how `isequal` and `==` differ when used with handles.

Represent State with Enumerations

The `MachineState` class defines two enumeration members to represent the state of a machine, either running or not running.

```
classdef MachineState
   enumeration
      Running
      NotRunning
   end
end
```

The `Machine` class represents a machine with start and stop operations. The `MachineState` enumerations are easy to work with because of their `eq` and `char` methods, and they result in code that is easy to read.

```
classdef Machine < handle
   properties (SetAccess = private)
```

```
      State = MachineState.NotRunning
   end

   methods
      function start(machine)
         if machine.State == MachineState.NotRunning
            machine.State = MachineState.Running;
         end
         disp (machine.State.char)
      end
      function stop(machine)
         if machine.State == MachineState.Running
            machine.State = MachineState.NotRunning;
         end
         disp (machine.State.char)
      end
   end
end
```

Create a `Machine` object and call `start` and `stop` methods

```
m = Machine;
m.start
```

```
Running
```

```
m.stop
```

```
NotRunning
```

See Also

Related Examples

- “Enumerations That Encapsulate Data” on page 14-37

Enumerations That Encapsulate Data

In this section...

Enumeration Classes with Properties

Enumeration classes can define properties to store data values. The enumeration members represent specific values for these properties, which MATLAB assigns in the class constructor. For information on defining enumeration classes, see “Define Enumeration Classes” on page 14-4.

Store Data in Properties

Note Enumeration classes that subclass built-in numeric or logical classes cannot define or inherit properties. For more information on this kind of enumeration class, see “Enumerations Derived from Built-In Classes” on page 14-27 .

Define properties in an enumeration class if you want to associate specific data with enumeration members, but do not need to inherit arithmetic, ordering, or other operations that MATLAB defines for specific built-in classes.

Representing Colors

Define an enumeration class to represent the RGB values of the colors in a color set. The `Colors` class defines names for the colors, each of which uses the RGB values as arguments to the class constructor:

```
classdef Colors
   properties
      R = 0
      G = 0
      B = 0
   end
   methods
      function c = Colors(r, g, b)
         c.R = r; c.G = g; c.B = b;
      end
   end
   enumeration
      Blueish   (18/255,104/255,179/255)
      Reddish   (237/255,36/255,38/255)
      Greenish  (155/255,190/255,61/255)
      Purplish  (123/255,45/255,116/255)
      Yellowish (1,199/255,0)
      LightBlue (77/255,190/255,238/255)
   end
end
```

You can access the property values via the enumeration member:

```
Colors.Reddish.R
```

```
ans =

    0.9294
```

Suppose that you want to create a plot with the new shade of red named `Reddish`:

```
a = Colors.Reddish;
[a.R,a.G,a.B]

ans =

    0.9294    0.1412    0.1490
```

Use these values by accessing the enumeration member properties. For example, the `myPlot` function accepts a `Colors` enumeration member as an input argument. The function accesses the RGB values defining the color from the property values.

```
function h = myPlot(x,y,LineColor)
   h = line('XData',x,'YData',y);
   r = LineColor.R;
   g = LineColor.G;
   b = LineColor.B;
   h.Color = [r g b];
end
```

Create a plot using a reddish color line:

```
h = myPlot(1:10,1:10,Colors.Reddish);
```

The `Colors` class encapsulates the definitions of a standard set of colors. You can change the enumeration class definition of the colors and not affect functions that use the enumerations.

Enumerations Defining Categories

The `Cars` class defines categories used to inventory automobiles. The `Cars` class derives from the `CarPainter` class, which derives from `handle`. The abstract `CarPainter` class defines a `paint` method, which modifies the `Color` property when a car is painted another color.

The `Cars` class uses the `Colors` enumeration members to specify a finite set of available colors. The exact definition of any given color can change independently of the `Cars` class.

```
classdef Cars < CarPainter
   enumeration
      Hybrid (2,'Manual',55,Colors.Reddish)
      Compact(4,'Manual',32,Colors.Greenish)
      MiniVan(6,'Automatic',24,Colors.Blueish)
      SUV    (8,'Automatic',12,Colors.Yellowish)
   end
   properties  (SetAccess = private)
      Cylinders
      Transmission
      MPG
      Color
   end
   methods
      function obj = Cars(cyl,trans,mpg,colr)
         obj.Cylinders = cyl;
         obj.Transmission = trans;
```

```
            obj.MPG = mpg;
            obj.Color = colr;
        end
        function paint(obj,colorobj)
            if isa(colorobj,'Colors')
                obj.Color = colorobj;
            else
                [~,cls] = enumeration('Colors');
                disp('Not an available color')
                disp(cls)
            end
        end
    end
end
```

The `CarPainter` class requires its subclasses to define a method called `paint`:

```
classdef CarPainter < handle
    methods (Abstract)
        paint(carobj,colorobj)
    end
end
```

Define an instance of the `Cars` class:

```
c1 = Cars.Compact;
```

The color of this car is `Greenish`, as defined by the `Colors.Greenish` enumeration:

```
c1.Color
```

```
ans =

    Greenish
```

Use the `paint` method to change the car color:

```
c1.paint(Colors.Reddish)
c1.Color
```

```
ans =

    Reddish
```

See Also

Related Examples

- “Save and Load Enumerations” on page 14-40
- “Enumerations for Property Values” on page 14-14

Save and Load Enumerations

In this section...

Basic Knowledge

See the `save` and `load` functions and “Save and Load Process for Objects” on page 13-2 for general information on saving and loading objects.

To see a list of enumeration names defined by a class, use the `enumeration` function.

Built-In and Value-Based Enumeration Classes

When you save enumerations that derive from built-in classes or that are value-based classes with properties, MATLAB saves the names of the enumeration members and the definition of each member.

When loading these enumerations, MATLAB preserves names over underlying values. If the saved named value is different from the current class definition, MATLAB uses the value defined in the current class, and then issues a warning.

Simple and Handle-Based Enumeration Classes

When you save simple enumerations that have no properties, superclasses, or values associated with the member names or enumerations derived from the `handle` class, MATLAB saves the names and any underlying values.

When loading these types of enumerations, MATLAB does not check the values associated with the names in the current class definition. This behavior results from the fact that simple enumerations have no underlying values and handle-based enumerations can legally have values that are different than those values defined by the class.

Causes: Load as struct Instead of Object

If you add a new named value or a new property to a class after saving an enumeration, MATLAB does not warn during load.

If the changes to the enumeration class definition do not prevent MATLAB from loading the object (that is, all the named values in the MAT-File are present in the modified class definition), then MATLAB issues a warning that the class has changed and loads the enumeration.

In these five cases, MATLAB issues a warning. In case 1, there are no defined results. In cases 2 through 5, MATLAB loads as much of the saved data as possible as a `struct`:

1 MATLAB cannot find the class definition.

2 The class is no longer an enumeration class.

3 MATLAB cannot initialize the class.

4 There are one or more enumeration members in the loaded enumeration that is not in the class definition.

5 If the class is a value-based enumeration with properties and a property that exists in the file, is not present in the class definition.

struct Fields

In cases 2 through 5, the returned `struct` has these fields:

- `ValueNames` — A cell array of strings, one per unique value in the enumeration array.
- `Values` — An array of the same dimension as `ValueNames` containing the corresponding values of the enumeration members named in `ValueNames`. Depending on the kind of enumeration class, `Values` can be one of these:
  - If the enumeration class derives from a built-in class, the array class is the same as the built-in class. The values in the array are the underlying values of each enumeration member.
  - Otherwise, a `struct` array representing the property name — property values pairs of each enumeration member. For simple and handle-based enumerations, the `struct` array has no fields.
- `ValueIndices` — a `uint32` array of the same size as the original enumeration. Each element is an index into the `ValueNames` and `Values` arrays. The content of `ValueIndices` represents the value of each object in the original enumeration array.

See Also

More About

- “Named Values” on page 14-2

Constant Properties

Define Class Properties with Constant Values

In this section...

Defining Named Constants

You can define constants that you can refer to by name by creating a MATLAB class that defines constant properties.

Use constant properties to define constant values that you can access by name. Create a class with constant properties by declaring the `Constant` attribute in the property blocks. Setting the `Constant` attribute means that, once initialized to the value specified in the property block, the value cannot be changed.

Assigning Values to Constant Properties

Assign any value to a `Constant` property, including a MATLAB expression. For example:

```
classdef NamedConst
   properties (Constant)
      R = pi/180
      D = 1/NamedConst.R
      AccCode = '0145968740001110202NPQ'
      RN = rand(5)
   end
end
```

MATLAB evaluates the expressions when loading the class. Therefore, the values MATLAB assigns to `RN` are the result of a single call to the `rand` function and do not change with subsequent references to `NamedConst.RN`. Calling `clear classes` causes MATLAB to reload the class and reinitialize the constant properties.

Referencing Constant Properties

Refer to the constant using the class name and the property name:

`ClassName.PropName`

For example, to use the `NamedConst` class defined in the previous section, reference the constant for the degree to radian conversion, `R`:

```
radi = 45*NamedConst.R
```

```
radi =

    0.7854
```

Constants in Packages

To create a library for constant values that you can access by name, first create a package folder, then define the various classes to organize the constants. For example, to implement a set of constants that are useful for making astronomical calculations, define a `AstroConstants` class in a package called `constants`:

```
+constants/@AstroConstants/AstroConstants.m
```

The class defines a set of `Constant` properties with values assigned:

```
classdef AstroConstants
   properties (Constant)
   C = 2.99792458e8     % m/s
   G = 6.67259          % m/kgs
   Me = 5.976e24        % Earth mass (kg)
   Re = 6.378e6         % Earth radius (m)
   end
end
```

To use this set of constants, reference them with a fully qualified class name. For example, the following function uses some of the constants defined in `AstroContants`:

```
function E = energyToOrbit(m,r)
   E = constants.AstroConstants.G * constants.AstroConstants.Me * m * ...
      (1/constants.AstroConstants.Re-0.5*r);
end
```

Importing the package into the function eliminates the need to repeat the package name (see `import`):

```
function E = energyToOrbit(m,r)
   import constants.*;
   E = AstroConstants.G * AstroConstants.Me * m * ...
      (1/AstroConstants.Re - 0.5 * r);
end
```

Constant Property Assigned a Handle Object

If a class defines a constant property with a value that is a handle object, you can assign values to the handle object properties. To access the handle object, create a local variable.

For example, the `ConstMapClass` class defines a constant property. The value of the constant property is a handle object (a `containers.Map` object).

```
classdef ConstMapClass < handle
   properties (Constant)
      ConstMapProp = containers.Map
   end
end
```

To assign the current date to the `Date` key, return the handle from the constant property, then make the assignment using the local variable on the left side of the assignment statement:

```
localMap = ConstMapClass.ConstMapProp
localMap('Date') = datestr(clock);
```

You cannot use a reference to a constant property on the left side of an assignment statement. For example, MATLAB interprets the following statement as the creation of a `struct` named `ConstMapClass` with a field `ConstMapProp`:

```
ConstMapClass.ConstMapProp('Date') = datestr(clock);
```

Constant Property Assigned Any Object

You can assign an instance of the defining class to a constant property. MATLAB creates the instance assigned to the constant property when loading the class. Use this technique only when the defining class is a `handle` class.

The `MyProject` is an example of such a class:

```
classdef MyProject < handle
   properties (Constant)
      ProjectInfo = MyProject
   end
   properties
      Date
      Department
      ProjectNumber
   end
   methods (Access = private)
      function obj = MyProject
         obj.Date = datestr(clock);
         obj.Department = 'Engineering';
         obj.ProjectNumber = 'P29.367';
      end
   end
end
```

Reference property data via the `Constant` property:

```
MyProject.ProjectInfo.Date
```

```
ans =

18-Apr-2002 09:56:59
```

Because `MyProject` is a handle class, you can get the handle to the instance that is assigned to the constant property:

```
p = MyProject.ProjectInfo;
```

Access the data in the `MyProject` class using this handle:

```
p.Department
```

```
ans =

Engineering
```

Modify the nonconstant properties of the `MyProject` class using this handle:

```
p.Department = 'Quality Assurance';
```

`p` is a handle to the instance of `MyProject` that is assigned to the `ProjectInfo` constant property:

```
MyProject.ProjectInfo.Department

ans =

Quality Assurance
```

Clearing the class results in the assignment of a new instance of `MyProject` to the `ProjectInfo` property.

```
clear MyProject
MyProject.ProjectInfo.Department

ans =

Engineering
```

You can assign an instance of the defining class as the default value of a property only when the property is declared as `Constant`

Constant Properties — No Support for Get Events

Constant properties do not support property `PreGet` or `PostGet` events. MATLAB issues a warning during class initialization if you set the `GetObservable` attribute of a `Constant` property to `true`.

See Also

Related Examples

- “Static Data” on page 4-2

More About

- “Named Values” on page 14-2

Information from Class Metadata

Class Metadata

In this section...

What Is Class Metadata?

Class metadata is information about class definitions that is available from various metaclasses objects. Use metaclass objects to obtain information without having to create instances of the class. Metadata enables the programmatic inspection of classes. Each metaclass has properties, methods, and events that contain information about the class or class component it describes.

All class components have an associated metaclass, which you access from the `meta.class` object. For example, create the `meta.class` object for the `matlab.alias.AliasFileManager` class:

```
mc = ?matlab.alias.AliasFileManager
```

```
mc =

  class with properties:

                    Name: 'matlab.alias.AliasFileManager'
             Description: 'AliasFileManager Create and edit alias definition files'
     DetailedDescription: '  Use an AliasFileManager object to create and manage...
                  Hidden: 0
                  Sealed: 1
                Abstract: 0
             Enumeration: 0
          ConstructOnLoad: 0
         HandleCompatible: 1
           InferiorClasses: {0×1 cell}
        ContainingPackage: [1×1 meta.package]
                  Aliases: [0×1 string]
    RestrictsSubclassing: 1
            PropertyList: [1×1 meta.property]
              MethodList: [28×1 meta.method]
               EventList: [1×1 meta.event]
   EnumerationMemberList: [0×1 meta.EnumeratedValue]
          SuperclassList: [1×1 meta.class]
```

The meta Package

The `meta` package contains metaclasses that describe the definition of classes and class components. The class name indicates the component described by the metaclass. For example, each class property has a meta.property associated with it. Attributes defined for class components correspond to properties in the respective metaclass object.

- `meta.package` — Access from `meta.class` `ContainingPackage` property.
- `meta.class` — Create from class name or class object using `metaclass` function or `?` operator.
- `meta.property` — Access from `meta.class` `PropertyList` property.
- `meta.DynamicProperty` — Obtain from the `addprop` method.

- `meta.method` — Access from `meta.class` `MethodList` property.
- `meta.event` — Access from `meta.class` `EventList` property.
- `meta.EnumeratedValue` — Access from `meta.class` `EnumerationMemberListList` property.

Metaclass Objects

You cannot instantiate metaclasses directly by calling the respective class constructor. Create metaclass objects from class instances or from the class name.

- `?`*`ClassName`* — Returns a `meta.class` object for the named class. Use `meta.class.fromName` with class names stored as characters in variables.
- `meta.class.fromName('`*`ClassName`*`')` — returns the `meta.class` object for the named class (`meta.class.fromName` is a `meta.class` method).
- `metaclass(obj)` — Returns a metaclass object for the class instance (`metaclass`)

Create `meta.class` object from class name using the `?` operator:

```
mc = ?MyClass;
```

Create `meta.class` object from class name using the `fromName` method:

```
mc = meta.class.fromName('MyClass');
```

Create `meta.class` object from class instance

```
obj = MyClass;
mc = metaclass(obj);
```

The `metaclass` function returns the `meta.class` object (that is, an object of the `meta.class` class). You can obtain other metaclass objects (`meta.property`, `meta.method`, and so on) from the `meta.class` object.

Note Metaclass is a term used here to refer to all the classes in the `meta` package. `meta.class` is a class in the `meta` package whose instances contain information about MATLAB classes. Metadata is information about classes contained in metaclasses.

Metaclass Object Lifecycle

When you change a class definition, MATLAB reloads the class definition. If instances of the class exist, MATLAB updates those objects according to the new definition.

However, MATLAB does not update existing metaclass objects to the new class definition. If you change a class definition while metaclass objects of that class exist, MATLAB deletes the metaclass objects and their handles become invalid. You must create a new metaclass object after updating the class.

For information on how to modify and reload classes, see “Automatic Updates for Modified Classes” on page 5-27.

See Also

Related Examples

- “Class Introspection with Metadata” on page 16-5
- “Find Objects with Specific Values” on page 16-9
- “Get Information About Properties” on page 16-12
- “Find Default Values in Property Metadata” on page 16-17

Class Introspection with Metadata

In this section...

Using Class Metadata

Use class metadata to get information about classes and objects programmatically. For example, you can determine attribute values for class members or get a list of events defined by the class. For basic information about metadata, see “Class Metadata” on page 16-2.

Inspect the EmployeeData Class

The `EmployeeData` class is a `handle` class with two properties, one of which has private `Access` and defines a set access method.

```
classdef EmployeeData < handle
   properties
      EmployeeName
   end
   properties (Access = private)
      EmployeeNumber
   end
   methods
      function obj = EmployeeData(name,ss)
         if nargin > 0
            obj.EmployeeName = name;
            obj.EmployeeNumber = ss;
         end
      end
      function set.EmployeeName(obj,name)
         if ischar(name)
            obj.EmployeeName = name;
         else
            error('Employee name must be a char vector')
         end
      end
   end
end
```

Inspect Class Definition

Using the `EmployeeData` class, create a `meta.class` object using the `?` operator:

```
mc = ?EmployeeData;
```

Determine from what classes `EmployeeData` derives. The returned value is a `meta.class` object for the `handle` superclass:

```
a = mc.SuperclassList;
a.Name
```

```
ans =

handle
```

The `EmployeeData` class has only one superclass. For classes having more than one direct superclass, `a` contains a `meta.class` object for each superclass.

Use an indexed reference to refer to any particular superclass:

```
a(1).Name
```

or, directly from `mc`:

```
mc.SuperclassList(1).Name
```

```
ans =

handle
```

The `SuperclassList` property contains only direct superclasses.

Inspect Properties

Find the names of the properties defined by the `EmployeeData` class. First obtain an array of `meta.properties` objects from the `meta.class` `PropertyList` property.

```
mc = ?EmployeeData;
mpArray = mc.PropertyList;
```

The length of `mpArray` indicates that there are two `meta.property` objects, one for each property defined by the `EmployeeData` class:

```
length(mpArray)
ans =
     2
```

Now get a `meta.property` object from the array:

```
prop1 = mpArray(1);
prop1.Name
```

```
ans =

EmployeeName
```

The `Name` property of the `meta.property` object identifies the class property represented by that `meta.property` object.

Query other `meta.property` object properties to determine the attributes of the `EmployeeName` properties.

Find Component with Specific Attribute

You can use indexing techniques to list class components that have specific attribute values. For example, this code lists the methods in the `EmployeeData` class that have `private` access:

```
mc = ?EmployeeData;
mc.PropertyList(ismember({mc.PropertyList(:).SetAccess},'private')).Name
```

```
ans =

EmployeeNumber
```

`Access` is not a property of the `meta.property` class. Use `SetAccess` and `GetAccess`, which are properties of the `meta.property` class.

Find components with attributes that are logical values using a statement like this one:

```
mc = ?handle;
mc.MethodList(ismember([mc.MethodList(:).Hidden],true)).Name
```

```
ans =

empty
```

Inspect Class Instance

Create an `EmployeeData` object and determine property access settings:

```
EdObj = EmployeeData('My Name',1234567);
mcEdObj = metaclass(EdObj);
mpArray = mcEdObj.PropertyList;
EdObj.(mpArray(1).Name) % Dynamic field names work with objects
```

The value of the `EmployeeName` property is the text `My Name`, which was assigned in the constructor.

```
ans =
   My Name
```

The value of the `EmployeeNumber` property is not accessible because the property has private `Access`.

```
EdObj.(mpArray(2).Name)
```

```
You cannot get the 'EmployeeNumber' property of EmployeeData.
```

```
mpArray(2).GetAccess
```

```
ans =
   private
```

Obtain a function `handle` to the `EmployeeName` property set access function:

```
mpArray(1).SetMethod
```

```
ans =
   @D:\MyDir\@EmployeeData\EmployeeData.m>EmployeeData.set.EmployeeName
```

Metaclass EnumeratedValues Property

The `meta.class` `EnumeratedValues` property contains an array of `meta.EnumeratedValue` objects, one for each enumeration member. Use the `meta.EnumeratedValue` `Name` property to obtain the enumeration member names defined by an enumeration class. For example, given the `WeekDays` enumeration class:

```
classdef WeekDays
   enumeration
      Monday, Tuesday, Wednesday, Thursday, Friday
```

```
    end
end
```

Query enumeration names from the `meta.class` object:

```
mc = ?WeekDays;
mc.EnumerationMemberList(2).Name
```

```
ans =

Tuesday
```

See Also

Related Examples

- “Find Objects with Specific Values” on page 16-9

Find Objects with Specific Values

In this section...

Find Handle Objects

Use the `handle` class `findobj` method to find objects that have properties with specific values. For example, the following class defines a `PhoneBook` object to represent a telephone book entry in a data base. The `PhoneBook` class subclasses the `dynamicprops` class, which derives from `handle`.

```
classdef PhoneBook < dynamicprops
   properties
      Name
      Address
      Number
   end
   methods
      function obj = PhoneBook(n,a,p)
         obj.Name = n;
         obj.Address = a;
         obj.Number = p;
      end
   end
end
```

Here are three of the `PhoneBook` entries in the database:

```
PB(1) = PhoneBook('Nancy Vidal','123 Washington Street','5081234567');
PB(2) = PhoneBook('Nancy Vidal','123 Main Street','5081234568');
PB(3) = PhoneBook('Nancy Wong','123 South Street','5081234569');
```

One of these three `PhoneBook` objects has a dynamic property:

```
PB(2).addprop('HighSpeedInternet');
PB(2).HighSpeedInternet = '1M';
```

Find Property/Value Pairs

Find the object representing employee `Nancy Wong` and display the name and number by concatenating the strings:

```
NW = findobj(PB,'Name','Nancy Wong');
[NW.Name,' - ',NW.Number]

ans =

Nancy Wong - 5081234569
```

Find Objects with Specific Property Names

Search for objects with specific property names using the `-property` option:

```
H = findobj(PB,'-property','HighSpeedInternet');
H.HighSpeedInternet
```

```
ans =

1M
```

The `-property` option enables you to omit the value of the property and search for objects using only the property name.

Using Logical Expressions

Search for specific combinations of property names and values:

```
H = findobj(PB,'Name','Nancy Vidal','-and','Address','123 Main Street');
H.Number
```

```
ans =

5081234568
```

Find by Attribute Settings

All metaclasses derive from the `handle` class. You can use the `handle` `findobj` method to find class members that have specific attribute settings.

For example, find the abstract methods in a class definition by searching the `meta.class` `MethodList` for `meta.method` objects with their `Abstract` property set to `true`:

Use the class name in character format because class is abstract. You cannot create an object of the class:

```
mc = meta.class.fromName('MyClass');
```

Search the `MethodList` list of `meta.method` objects for those methods that have their `Abstract` property set to `true`:

```
absMethods = findobj(mc.MethodList,'Abstract',true);
methodNames = {absMethods.Name};
```

The cell array, `methodNames`, contains the names of the abstract methods in the class.

Find Properties That Have Public Get Access

Find the names of all properties in the `containers.Map` class that have public `GetAccess`:

- Get the `meta.class` object.
- Use `findobj` to search the array of `meta.property` objects.
- Use braces to create a cell array of property names.

```
mc = ?containers.Map;
mpArray = findobj(mc.PropertyList,'GetAccess','public');
names = {mpArray.Name};
```

Display the names of all `containers.Map` properties that have public `GetAccess`:

```
celldisp(names)
```

```
names{1} =
```

```
Count

 names{2} =

KeyType

names{3} =

ValueType
```

Find Static Methods

Determine if any `containers.Map` class methods are static:

```
~isempty(findobj([mc.MethodList(:)],'Static',true))

ans =

     1
```

`findobj` returns an array of `meta.method` objects for the static methods. In this case, the list of static methods is not empty. Therefore, there are static methods defined by this class.

Get the names of any static methods from the `meta.method` array:

```
staticMethodInfo = findobj([mc.MethodList(:)],'Static',true);
staticMethodInfo(:).Name

ans =

empty
```

The name of the static method (there is only one in this case) is `empty`. Here is the information from the `meta.method` object for the `empty` method:

```
staticMethodInfo

  method with properties:

                  Name: 'empty'
           Description: 'Returns an empty object array of the given size'
    DetailedDescription: ''
                 Access: 'public'
                 Static: 1
               Abstract: 0
                 Sealed: 0
                 Hidden: 1
             InputNames: {'varargin'}
            OutputNames: {'E'}
          DefiningClass: [1x1 meta.class]
```

See Also

`empty`

Related Examples

- “Get Information About Properties” on page 16-12

Get Information About Properties

In this section...

The meta.property Object

Use the `meta.property` class to determine the values of property attributes. The writable properties of a `meta.property` object correspond to the attributes of the associated property. The values of the writable `meta.property` properties correspond to the attribute values specified in the class definition.

You can get the `meta.property` object for a property from the `meta.class` object. To get the `meta.class` object for a class:

- Use the `metaclass` function on an object of the class.
- Use the `?` operator with the class name.

For example, the `BasicHandle` class defines three properties:

```
classdef BasicHandle < handle
   % BasicHandle  Inherits from handle superclass
   % Defines 1 public and 2 private properties.
   properties (SetAccess = private)
      Date = date
      PassKey = randi(9,[1,7])
   end
   properties
      Category {mustBeMember(Category,{'new','change'})} = 'new'
   end
end
```

Create the `meta.class` object using the `?` operator with the class name:

```
mc = ?BasicHandle
```

```
mc =

  class with properties:

                    Name: 'BasicHandle'
             Description: 'BasicHandle  Inherits from handle superclass'
       DetailedDescription: '  Defines 1 public and 2 private properties.'
                  Hidden: 0
                  Sealed: 0
                Abstract: 0
             Enumeration: 0
          ConstructOnLoad: 0
         HandleCompatible: 1
          InferiorClasses: {0×1 cell}
        ContainingPackage: [0×0 meta.package]
                  Aliases: [0×1 string]
      RestrictsSubclassing: 0
```

```
           PropertyList: [3×1 meta.property]
             MethodList: [24×1 meta.method]
              EventList: [1×1 meta.event]
   EnumerationMemberList: [0×1 meta.EnumeratedValue]
          SuperclassList: [1×1 meta.class]
```

The `meta.class` object property named `PropertyList` contains an array of `meta.property` objects, one for each property defined by the class. For example, the name of the property associated with the `meta.property` object in element `1` is:

```
mc.PropertyList(1).Name

ans =

Date
```

The `meta.class` object contains a `meta.property` object for all properties, including hidden properties. The `properties` function returns only public properties.

For a handle class, use the `handle` `findprop` method to get the `meta.property` object for a specific property.

For example, find the `meta.property` object for the `Category` property of the `BasicHandle` class.

```
h = BasicHandle;
mp = findprop(h,'Category')

mp =

  property with properties:

                  Name: 'Category'
           Description: ''
    DetailedDescription: ''
             GetAccess: 'public'
             SetAccess: 'public'
             Dependent: 0
              Constant: 0
              Abstract: 0
             Transient: 0
                Hidden: 0
         GetObservable: 0
         SetObservable: 0
              AbortSet: 0
           NonCopyable: 0
             GetMethod: []
             SetMethod: []
            HasDefault: 1
          DefaultValue: 'new'
         DefiningClass: [1×1 meta.class]
```

The `meta.property` display shows that a default `BasicHandle` object `Category` property:

- Has public `GetAccess` and `SetAccess`
- Has a default value of `new`

For a list of property attributes, see “Table of Property Attributes” on page 8-8.

How to Index Metaclass Objects

Access other metaclass objects directly from the `meta.class` object properties. For example, the statement:

```
mc = ?containers.Map;
```

returns a `meta.class` object:

```
class(mc)
```

```
ans =

meta.class
```

Referencing the `PropertyList` `meta.class` property returns an array with one `meta.property` object for each property of the `containers.Map` class:

```
class(mc.PropertyList)
```

```
ans =

meta.property
```

Each array element is a single `meta.property` object:

```
mc.Properties(1)
```

```
ans =

    [1x1 meta.property]
```

The `Name` property of the `meta.property` object contains a `char` vector that is the name of the property:

```
class(mc.PropertyList(1).Name)
```

```
ans =

char
```

Apply standard MATLAB indexing to access information in metaclass objects.

For example, the `meta.class` `PropertyList` property contains an array of `meta.property` objects. The following expression accesses the first `meta.property` object in this array and returns the first and last letters (`C` and `t`) of the `char` vector contained in the `meta.property` `Name` property.

```
mc.PropertyList(1).Name([1 end])
```

```
ans =

Ct
```

How to Find Properties with Specific Attributes

This example implements a function that finds properties with specific attribute values. For example, you can:

- Find objects that define constant properties (`Constant` attribute set to `true`).
- Determine what properties are read-only (`GetAccess = public`, `SetAccess = private`).

The `findAttrValue` function returns a cell array of property names that set the specified attribute. The function accesses information from metadata using these techniques:

- If input argument, `obj`, is a `char` vector, use the `meta.class.fromName` static method to get the `meta.class` object.
- If input argument, `obj`, is an object, use the `metaclass` function to get the `meta.class` object.
- Every property has an associated `meta.property` object. Obtain these objects from the `meta.class` `PropertyList` property.
- Use the `handle` class `findprop` method to determine if the requested property attribute is a valid attribute name. All property attributes are properties of the `meta.property` object. The statement, `findobj(mp,'PropertyName')` determines whether the `meta.property` object, `mp`, has a property called `PropertyName`.
- Reference `meta.property` object properties using dynamic field names. For example, if `attrName = 'Constant'`, then MATLAB converts the expression `mp.(attrName)` to `mp.Constant`
- The optional third argument enables you to specify the value of attributes whose values are not logical `true` or `false` (such as `GetAccess` and `SetAccess`).

```
function cl_out = findAttrValue(obj,attrName,varargin)
   if ischar(obj)
      mc = meta.class.fromName(obj);
   elseif isobject(obj)
      mc = metaclass(obj);
   end
   ii = 0; numb_props = length(mc.PropertyList);
   cl_array = cell(1,numb_props);
   for  c = 1:numb_props
      mp = mc.PropertyList(c);
      if isempty (findprop(mp,attrName))
         error('Not a valid attribute name')
      end
      attrValue = mp.(attrName);
      if attrValue
         if islogical(attrValue) || strcmp(varargin{1},attrValue)
            ii = ii + 1;
            cl_array(ii) = {mp.Name};
         end
      end
   end
   cl_out = cl_array(1:ii);
end
```

Find Property Attributes

Define a `containers.Map` object:

```
mapobj = containers.Map({'rose','bicycle'},{'flower','machine'});
```

Find properties with `private` `SetAccess`:

```
findAttrValue(mapobj,'SetAccess','private')
```

```
ans =

    'Count'    'KeyType'    'ValueType'    'serialization'
```

Find properties with `public` `GetAccess`:

```
findAttrValue(mapobj,'GetAccess','public')
```

```
ans =

    'Count'    'KeyType'    'ValueType'
```

See Also

Related Examples

- “Find Default Values in Property Metadata” on page 16-17

Find Default Values in Property Metadata

In this section...

Default Values

Class definitions can specify explicit default values for properties (see “Define Properties with Default Values” on page 8-13). Determine if a class defines an explicit default value for a property and what the value of the default is from the property `meta.property` object.

meta.property Data

The `meta.class` object for a class contains a `meta.property` object for every property defined by the class, including properties with private and protected access.

For example, get the `meta.class` object for the `PropertyWithDefault` class shown here:

```
classdef PropertyWithDefault
   properties
      Date = date
      RandNumber = randi(9)
   end
end
```

Get an array of `meta.property` objects from the `meta.class` object:

```
mc = ?PropertyWithDefault; % meta.class object
mp = mc.PropertyList; % meta.property array
```

The second element in the `mp` array is the `meta.property` object for the `RandNumber` property. Listing the `meta.property` object shows the information contained in its properties:

```
mp(2)
```

```
  property with properties:

                  Name: 'RandNumber'
           Description: ''
    DetailedDescription: ''
             GetAccess: 'public'
             SetAccess: 'public'
             Dependent: 0
              Constant: 0
              Abstract: 0
             Transient: 0
                Hidden: 0
         GetObservable: 0
         SetObservable: 0
              AbortSet: 0
           NonCopyable: 0
             GetMethod: []
             SetMethod: []
```

```
      HasDefault: 1
    DefaultValue: 5
   DefiningClass: [1×1 meta.class]
```

Two of the listed `meta.property` properties provide information on default values:

- `HasDefault` — `true` (displayed as 1) if the class specifies a default value for the property, `false` if it does not.
- `DefaultValue` — Contains the default value, when the class defines a default value for the property. If the default value is an expression, the value of `DefaultValue` is the result of evaluating the expression.

For more information on the evaluation of property default values defined by expressions, see “Evaluation of Expressions in Class Definitions” on page 6-9.

These properties provide a programmatic way to obtain property default values without opening class definition files. Use these `meta.property` object properties to obtain property default values for both built-in classes and classes defined in MATLAB code.

Query Default Value

The procedure for querying a default value involves:

1. Getting the `meta.property` object for the property whose default value you want to query.
2. Testing the logical value of the `meta.property` `HasDefault` property to determine if the property defines a default value. MATLAB returns an error when you query the `DefaultValue` property if the class does not define a default value for the property.
3. Obtaining the default value from the `meta.property` `DefaultValue` property if the `HasDefault` value is `true`.

Use the `?` operator, the `metaclass` function, or the `meta.class.fromName` static method (works with `char` vector variable) to obtain a `meta.class` object.

The `meta.class` object `PropertyList` contains an array of `meta.property` objects. Identify which property corresponds to which `meta.property` object using the `meta.property` `Name` property.

For example, this class defines properties with default values:

```
classdef MyDefs
   properties
      Material = 'acrylic'
      InitialValue = 1.0
   end
end
```

Follow these steps to obtain the default value defined for the `Material` property. Include any error checking that is necessary for your application.

1. Get the `meta.class` object for the class:

   ```
   mc = ?MyDefs;
   ```
2. Get an array of `meta.property` objects from the `meta.class` `PropertyList` property:

   ```
   mp = mc.PropertyList;
   ```
3. The length of the `mp` array equals the number of properties. You can use the `meta.property` `Name` property to find the property of interest:

```
for k = 1:length(mp)
   if (strcmp(mp(k).Name,'Material')
      ...
```

4 Before querying the default value of the `Material` property, test the `HasDefault` `meta.property` to determine if `MyClass` defines a default property for this property:

```
if mp(k).HasDefault
         dv = mp(k).DefaultValue;
      end
```

The `DefaultValue` property is read-only. Changing the default value in the class definition changes the value of `DefaultValue` property. You can query the default value of a property regardless of its access settings.

Abstract and dynamic properties cannot define default values. Therefore, MATLAB returns an error if you attempt to query the default value of properties with these attributes. Always test the logical value of the `meta.property` `HasDefault` property before querying the `DefaultValue` property to avoid generating an error.

Default Values Defined as Expressions

Class definitions can define property default values as MATLAB expressions (see “Evaluation of Expressions in Class Definitions” on page 6-9 for more information). MATLAB evaluates these expressions the first time the default value is needed, such as the first time you create an instance of the class.

Querying the `meta.property` `DefaultValue` property causes MATLAB to evaluate a default value expression, if it had not yet been evaluated. Therefore, querying a property default value can return an error or warning if errors or warnings occur when MATLAB evaluates the expression. See “Property with Expression That Errors” on page 16-20 for an example.

Property with No Explicit Default Value

`MyClass` does not explicitly define a default value for the `Foo` property:

```
classdef MyFoo
   properties
      Foo
   end
end
```

The `meta.property` instance for property `Foo` has a value of `false` for `HasDefault`. Because the class does not explicitly define a default value for `Foo`, attempting to access the `DefaultValue` property causes an error:

```
mc = ?MyFoo;
mp = mc.PropertyList(1);
mp.HasDefault

ans =

    0

dv = mp.DefaultValue;

No default value has been defined for property Foo
```

Abstract Property

`MyClass` defines the `Foo` property as `Abstract`:

```
classdef MyAbst
   properties (Abstract)
      Foo
   end
end
```

The `meta.property` instance for property `Foo` has a value of `false` for its `HasDefault` property because you cannot define a default value for an `Abstract` property. Attempting to access `DefaultValue` causes an error:

```
mc = ?MyAbst;
mp = mc.PropertyList(1);
mp.HasDefault
```

```
ans =

    0
```

```
dv = mp.DefaultValue;
```

```
Property Foo is abstract and therefore cannot have a default value.
```

Property with Expression That Errors

`MyPropEr` defines the `Foo` property default value as an expression that errors when evaluated.

```
classdef MyPropEr
   properties
      Foo = sin(pie/2)
   end
end
```

The `meta.property` object for property `Foo` has a value of `true` for its `HasDefault` property because `Foo` does have a default value:

```
sin(pie/2)
```

However, this expression returns an error (`pie` is a function that creates a pie graph, not the value `pi`).

```
mc = ?MyPropEr;
mp = mc.PropertyList(1);
mp.HasDefault
```

```
ans =

    1
```

```
dv = mp.DefaultValue;
```

```
Error using pie (line 29)
Not enough input arguments.
```

Querying the default value causes the evaluation of the expression and returns the error.

Property With Explicitly Defined Default Value of Empty

`MyEmptyProp` assigns a default of `[]` (empty double) to the `Foo` property:

```
classdef MyEmptyProp
   properties
      Foo = []
   end
end
```

The `meta.property` object for property `Foo` has a value of true for its `HasDefault` property. Accessing `DefaultValue` returns the value `[]`:

```
mc = ?MyEmptyProp;
mp = mc.PropertyList(1);
mp.HasDefault

ans =

    1

dv = mp.DefaultValue;

dv =

    []
```

See Also

Related Examples

- “Get Information About Properties” on page 16-12

Specialize Object Behavior

Methods That Modify Default Behavior

In this section...

How to Customize Class Behavior

There are functions that MATLAB calls implicitly when you perform certain actions with objects. For example, a statement like `[B(1);A(3)]` involves indexed reference and vertical concatenation.

You can change how user-defined objects behave by defining methods that control specific behaviors. To change a behavior, implement the appropriate method with the name and signature of the MATLAB function.

Which Methods Control Which Behaviors

The following table lists the methods to implement for your class and describes the behaviors that they control.

Class Method to Implement	Description
Concatenating Objects	
`cat`, `horzcat`, and `vertcat`	Customize behavior when concatenating objects See “Subclasses of Built-In Types with Properties” on page 12-53.
Creating Empty Arrays	
`empty`	Create empty arrays of the specified class. See “Empty Arrays” on page 10-7.
Displaying Objects	

`display`
 	
`disp`
	MATLAB calls `display` when an expression or statement is not terminated by a semicolon. This displays the result and any relevant variable names.
	
	`disp(E)` displays only the result of the expression. Overloading `disp` is one way to customize how your objects are displayed.
	
	See “Overloading the disp Function” on page 18-34.

Converting Objects to Other Classes	
converters like `double` and `char`	Convert an object to a MATLAB built-in class See “The Character Converter” on page 19-10 and “The Double Converter” on page 19-10.
Saving and Loading Objects	

Class Method to Implement	Description
`loadobj` and `saveobj`	Customize behavior when loading and saving objects See “Object Save and Load”.
Reshape and Rearrange	
`permute`	Rearrange dimensions of N-D array
`transpose`	Transpose vector or matrix
`ctranspose`	Complex conjugate transpose
`reshape`	Reshape array
`repmat`	Replicate array along specified dimensions
Determine Size and Shape	
`isscalar`	Determine if the input is a scalar
`isvector`	Determine if the input is a vector
`ismatrix`	Determine if the input is a matrix
`isempty`	Determine if the input is empty

Overload Functions and Override Methods

Overloading and overriding are terms that describe techniques for customizing class behavior. Here is how we use these terms in MATLAB.

Overloading

Overloading means that there is more than one function or method having the same name within the same scope. MATLAB dispatches to a particular function or method based on the dominant argument. For example, the `timeseries` class overloads the MATLAB `plot` function. When you call `plot` with a `timeseries` object as an input argument, MATLAB calls the `timeseries` class method named `plot`.

To call the nonoverloaded function, use the `builtin` function.

Overriding

Overriding means redefining a method inherited from a superclass. MATLAB dispatches to the most specific version of the method. That is, if the dominant argument is an object of the subclass, then MATLAB calls the subclass method.

To control class dominance, use the `InferiorClasses` attribute.

See Also

Related Examples

- “Overload Functions in Class Definitions” on page 9-25
- “Method Invocation” on page 9-11
- “Operator Overloading” on page 17-19

Concatenation Methods

In this section...

Default Concatenation

You can concatenate objects into arrays. For example, suppose that you have three instances of the class `MyClass`, `obj1`, `obj2`, `obj3`. You can form arrays of these objects using brackets. Horizontal concatenation calls `horzcat`:

```
HorArray = [obj1,obj2,obj3];
```

`HorArray` is a 1-by-3 array of class `MyClass`. You can concatenate the objects along the vertical dimension, which calls `vertcat`:

```
VertArray = [obj1;obj2;obj3]
```

`VertArray` is a 3-by-1 array of class `MyClass`. To concatenate arrays along different dimensions, use the `cat` function. For example:

```
ndArray = cat(3,HorArray,HorArray);
```

`ndArray` is a 1-by-3-by-2 array.

Methods to Overload

Overload `horzcat`, `vertcat`, and `cat` to produce specialized behaviors in your class. Overload both `horzcat` and `vertcat` whenever you want to modify object concatenation because MATLAB uses both functions for any concatenation operation.

See Also

Related Examples

- “Subclasses of Built-In Types with Properties” on page 12-53

Object Converters

In this section...

Why Implement Converters

You can convert an object of one class to an object of another class. A converter method has the same name as the class it converts to, such as `char` or `double`. Think of a converter method as an overloaded constructor method of another class. The converter takes an instance of its own class and returns an object of a different class.

Converters enable you to:

- Use methods defined for another class
- Ensure that expressions involving objects of mixed class types execute properly
- Control how instances are interpreted in other contexts

Suppose that you define a `polynomial` class. If you create a `double` method for the `polynomial` class, you can use it to call other functions that require inputs of type `double`.

```
p = polynomial(...);
dp = double(p);
roots(dp)
```

`p` is a polynomial object, `double` is a method of the `polynomial` class, and `roots` is a standard MATLAB function whose input arguments are the coefficients of a polynomial.

Converters for Package Classes

Classes defined in packages can have names that are a dot-separated list of names. The last name is a class and preceding names are packages. Name the conversion methods using the package qualifiers in the method names. For example, a conversion method to convert objects of `MyClass` to objects of the `PkgName.PkgClass` class uses this method name:

```
classdef MyClass
   ...
   methods
      function objPkgClass = PkgName.PkgClass(objMyclass)
         ...
      end
   end
end
```

You cannot define a converter method that uses dots in the name in a separate file. Define package-class converters in the `classdef` file.

Converters and Subscripted Assignment

When you make a subscripted assignment statement like:

```
A(1) = myobj;
```

MATLAB compares the class of the Right-Side variable to the class of the Left-Side variable. If the classes are different, MATLAB attempts to convert the Right-Side variable to the class of the Left-Side variable. To do this conversion, MATLAB first searches for a method of the Right-Side class that has the same name as the Left-Side class. Such a method is a converter method, which is similar to a typecast operation in other languages.

If the Right-Side class does not define a method to convert from the Right-Side class to the Left-Side class, MATLAB calls the Left-Side class constructor. passing it the Right-Side variable.

For example, suppose that you make the following assignments:

```
A(1) = objA;
A(2) = objB;
```

MATLAB attempts to call a method of `ClassB` named `ClassA`. If no such converter method exists, MATLAB software calls the `ClassA` constructor, passing `objB` as an argument. If the `ClassA` constructor cannot accept `objB` as an argument, then MATLAB returns an error.

Use `cell` arrays to store objects of different classes.

Converter for Heterogeneous Arrays

To support the formation of heterogeneous arrays using objects that are not part of the heterogeneous hierarchy, implement a `convertObject` method in the root superclass. The `convertObject` method must convert the nonmember object to a valid member of the heterogeneous hierarchy.

For details on implementing the `convertObject` method, see `matlab.mixin.Heterogeneous`.

See Also

Related Examples

- “Converter Methods” on page 10-19
- “The Double Converter” on page 19-10

Customize Object Indexing

In this section...

Default Object Indexing

MATLAB classes support object array indexing by default. Many class designs require no modification to this behavior.

Arrays enable you to reference and assign elements of the array using a subscripted notation. This notation specifies the indices of specific array elements. For example, suppose that you create two arrays of numbers (using `randi` and concatenation).

Create a 3-by-4 array of integers from 1 through 9:

```
A = randi(9,3,4)
```

```
A =

     4     8     5     7
     4     2     6     3
     7     5     7     7
```

Create a 1-by-3 array of the numbers 3, 6, 9:

```
B = [3 6 9];
```

Reference and assign elements of either array using index values in parentheses:

```
B(2) = A(3,4);
B
```

```
B =
     3     7     9
```

The MATLAB default behavior also works with user-defined objects. For example, create an array of objects of the same class:

```
for k=1:3
   objArray(k) = MyClass;
end
```

Referencing the second element in the object array, `objArray`, returns the object constructed when `k = 2`:

```
D = objArray(2);
class(D)
```

```
ans =

MyClass
```

You can assign an object to an array of objects of the same class, or an uninitialized variable:

```
newArray(3,4) = D;
```

Arrays of objects behave much like numeric arrays in MATLAB. You do not need to implement any special methods to provide standard array behavior with your class.

For general information about array indexing, see “Array Indexing”.

Customize Object Indexing With Modular Indexing Classes

Since R2021b. Recommended over “Code Patterns for subsref and subsasgn Methods” on page 17-9.

To modify indexing behavior for your class, inherit from one or more modular indexing mixin classes. Each class is responsible for one group of indexing operations:

- `matlab.mixin.indexing.RedefinesParen`—parentheses reference, assignment, and deletion
- `matlab.mixin.indexing.RedefinesDot`—dot property reference, method call, and assignment
- `matlab.mixin.indexing.RedefinesBrace`—brace reference and assignment

Each class defines abstract methods that handle the details of each indexing operation the class defines. Implement these methods to perform the operations your design requires.

You can inherit from these classes independently. For example, you can customize only parentheses indexing by inheriting only from `RedefinesParen`. The dot and brace indexing behaviors in that case are the default MATLAB behaviors.

You can also choose to customize just one or two levels of indexing and forward additional operations to another MATLAB object. For example, you can author a class that customizes parentheses indexing (using `RedefinesParen`) but uses the default behavior for dot method calls:

```
myInstance(2,1).value
```

See “Customize Parentheses Indexing” for an example of this behavior.

See Also

Related Examples

- “Customize Parentheses Indexing for Mapping Class” on page 17-23

Code Patterns for subsref and subsasgn Methods

In this section...

For R2021b and later, the recommended process for customizing indexing is to use the modular indexing superclasses instead of overloading `subsref` and `subsasgn`. For more information, see “Customize Object Indexing” on page 17-7.

Customize Indexed Reference and Assignment

User-defined classes have the same indexing behaviors as that of built-in classes. Classes can customize indexing operations by overloading the functions that MATLAB calls to evaluate indexing expressions. Overload the `subsref` and `subsasgn` functions when you want to define special behaviors for indexed reference and assignment.

Syntax for subsref and subsasgn Methods

MATLAB calls the `subsref` and `subsasgn` methods of your class with these arguments.

Method	Input	Output
`b = subsref(obj,s)`	• `obj` — Object or object array used in indexing expression • `s` — Indexing structure	`b` — Result of indexing expression
`obj = subsasgn(obj,s,b)`	• `obj` — Object or object array used in indexing expression • `s` — Indexing structure • `b` — Value being assigned	`obj` — Object or object array after assignment

Modifying Number of Arguments

If your class design requires that indexing operations return or assign a different number of values than the number defined by the default indexing operation, overload the `numArgumentsFromSubscript` function to control `nargout` for `subsref` and `nargin` for `subsasgn`. For more information and examples, see `numArgumentsFromSubscript`.

Indexing Structure Describes Indexing Expressions

The indexing structure contains information that describes the indexing expression. Class methods use the information in the indexing structure to evaluate the expression and implement custom behavior.

For example, the `CustomIndex` class defines a property that you can use in indexing expressions.

```
classdef CustomIndex
   properties
      DataArray
   end
end
```

Create an object and assign a 5-by-5 matrix created by the `magic` function to the `DataArray` property.

```
a = CustomIndex;
a.DataArray = magic(5);
```

This subscripted reference expression returns the first row of the 5-by-5 matrix.

```
a.DataArray(1,:)
```

```
ans =

    17    24     1     8    15
```

This expression assigns new values to the first row of the array stored in the `DataArray` property.

```
a.DataArray(1,:) = [1 2 3 4 5];
```

This assignment statement uses:

- A `'.'` type reference
- A property name following the dot (that is, `DataArray`)
- A range of indices (`1,:`) within parentheses

The indexing structure contains this information in the `type` and `subs` fields.

Values of the Indexing Structure

When executing an indexing expression, MATLAB calls the class `subsref` or `subsasgn` method, if the class overloads these functions. One of the arguments passed to the method is the indexing structure. The indexing structure has two fields:

- `type` — One of the three possible indexing types: `'.'`, `'()'`, `'{}'`
- `subs` — A `char` vector with the property name or cell array of the indices used in the expression, including `:` and `end`.

If the indexing expression is a compound expression, then MATLAB passes an array of structures, one `struct` for each level of indexing. For example, in this expression:

```
a.DataArray(1,:)
```

the indexing structure array `S` has these values:

- `S(1).type` is set to `'.'`, indicating that the first indexing operation is a dot.
- `s(1).subs` is set to the property name, `'DataArray'`

The second level of indexing is in the second element of the indexing structure:

- `S(2).types` is set to `'()'` indicating the second indexing operation is parentheses indexing

- `S(2).subs` is set to a cell array containing the indices `{[1],[:]}`

Typical Patterns for Indexing Methods

To overload the `subsref` and `subasgn` functions:

- Determine the full indexing expression using the `types` and `subs` fields of the indexing structure.
- Implement the specialized behaviors for the indexing operations supported by the class.
- Return the appropriate values or modified objects in response to the call by MATLAB.

A `switch` statement is a convenient way to detect the first level of indexing. There are three types of indexing—dot, parentheses, and braces. Each `case` block in the `switch` statement implements all indexing expressions that begin with that first-level type of indexing.

The methods must implement all indexing expressions that the class supports. If you do not customize a particular type of indexing, call the built-in function to handle that expression.

Use the length of the indexing structure array and indexing type define conditional statements for compound indexing expressions.

Code Framework for subsref Method

The following framework for the `subsref` method shows how to use information in the indexing structure in conditional statements. Your application can involve other expressions not shown here.

```
function varargout = subsref(obj,s)
   switch s(1).type
      case '.'
         if length(s) == 1
            % Implement obj.PropertyName
            ...
         elseif length(s) == 2 && strcmp(s(2).type,'()')
            % Implement obj.PropertyName(indices)
            ...
         else
            [varargout{1:nargout}] = builtin('subsref',obj,s);
         end
      case '()'
         if length(s) == 1
            % Implement obj(indices)
            ...
         elseif length(s) == 2 && strcmp(s(2).type,'.')
            % Implement obj(ind).PropertyName
            ...
         elseif length(s) == 3 && strcmp(s(2).type,'.') && strcmp(s(3).type,'()')
            % Implement obj(indices).PropertyName(indices)
            ...
         else
            % Use built-in for any other expression
            [varargout{1:nargout}] = builtin('subsref',obj,s);
         end
      case '{}'
         if length(s) == 1
            % Implement obj{indices}
            ...
         elseif length(s) == 2 && strcmp(s(2).type,'.')
            % Implement obj{indices}.PropertyName
            ...
         else
            % Use built-in for any other expression
            [varargout{1:nargout}] = builtin('subsref',obj,s);
         end
      otherwise
```

```
        error('Not a valid indexing expression')
    end
```

Using `varargout` for the returned value enables the method to work with object arrays. For example, suppose that you want to support the return of a comma-separated list with an expression like this one:

```
[x1,...xn] = objArray.PropertyName(Indices)
```

This expression results in a two-element indexing structure array. The first-level type is dot (`'.'`) and the second level is parentheses (`'()'`). Build the `varargout` cell array with each value in the array.

```
case '.'
   ...
   if length(s)==2 && strcmp(s(2).type,'()')
      prop = s(1).subs;       % Property name
      n = numel(obj);         % Number of elements in array
      varargout = cell(1,n); % Preallocate cell array
      for k = 1:n
         varargout{k} = obj(k).(prop).(s(2).subs);
      end
   end
   ...
end
```

subsasgn Pattern

The following framework for the `subsasgn` method shows how to use the indexing structure in conditional statements that implement assignment operations.

```
function obj = subsasgn(obj,s,varargin)

   % Allow subscripted assignment to uninitialized variable
   if isequal(obj,[])
      % obj = ClassName.empty;
   end

   switch s(1).type
      case '.'
         if length(s) == 1
            % Implement obj.PropertyName = varargin{:};
            ...
         elseif length(s) == 2 && strcmp(s(2).type,'()')
            % Implement obj.PropertyName(indices) = varargin{:};
            ...
         else
            % Call built-in for any other case
            obj = builtin('subsasgn',obj,s,varargin{:});
         end
      case '()'
         if length(s) == 1
            % Implement obj(indices) = varargin{:};
         elseif length(s) == 2 && strcmp(s(2).type,'.')
            % Implement obj(indices).PropertyName = varargin{:};
            ...
         elseif length(s) == 3 && strcmp(s(2).type,'.') && strcmp(s(3).type,'()')
            % Implement obj(indices).PropertyName(indices) = varargin{:};
            ...
         else
            % Use built-in for any other expression
            obj = builtin('subsasgn',obj,s,varargin{:});
         end
      case '{}'
         if length(s) == 1
            % Implement obj{indices} = varargin{:}
            ...
```

```
        elseif length(s) == 2 && strcmp(s(2).type,'.')
           % Implement obj{indices}.PropertyName = varargin{:}
           ...
           % Use built-in for any other expression
           obj = builtin('subsasgn',obj,s,varargin{:});
        end
     otherwise
        error('Not a valid indexing expression')
  end
end
```

Using `varargin` for the right-side value of the assignment statement enables the method to work with object arrays. For example, suppose that you want to support the assignment of a comma-separated list with an expression like this one:

```
C = {'one';'two';'three'};
[objArray.PropertyName] = C{:}
```

This expression results in an indexing structure with the dot type (`'.'`) indexing The cell array `C` on the right side of the assignment statement produces a comma-separated list. This code assigns one list item to each property in the object array.

```
case '.'
   if length(s)==1
      prop = s(1).subs;      % Property name
      n = numel(obj);        % Number of elements in array
      for k = 1:n
         obj(k).(prop) = varargin{k};
      end
   end
end
```

Subscripted Assignment with an Uninitialized Variable

Assigning to an element of an uninitialized variable results in a call to the `subsasgn` method of the class on the right side of the assignment. For example, this class defines a `subsasgn` method that simply calls the built-in `subsasgn` method for parenthesis indexing.

```
classdef MyClass
   methods
      function obj = subsasgn(obj,s,varargin)
         switch s(1).type
            case '()'
               obj = builtin('subsasgn',obj,s,varargin{:});
         end
      end
   end
end
```

When attempting to assign an object of `MyClass` to the first element of the uninitialized variable, `B(1)` in the following statement, MATLAB calls the `subsasgn` method of `MyClass` with an empty double (`[]`) as the first argument. The assignment can cause an error because the `subsasgn` method must be passed an object of the class.

```
clear B
B(1) = MyClass;
```

```
The following error occurred converting from MyClass to double:
Conversion to double from MyClass is not possible.
```

```
Error in MyClass/subsasgn (line 6)
            obj = builtin('subsasgn',obj,s,varargin{:});
```

The `subsasgn` method can detect this situation and take the appropriate action, such as returning a useful error message if the class does not support this type of assignment, or converting the input to an object of the class and passing it to `subsasgn`.

For example, because `MyClass` can allow subscripted assignment to an uninitialized variable, the `subsasgn` method can change the first argument from the empty double to an empty `MyClass` object.

Use the `isequal` function to check the input and the `empty` static method to create the empty object.

```
classdef MyClass
    methods
        function obj = subsasgn(obj,s,varargin)
            if isequal(obj,[])
                obj = MyClass.empty;
            end
            obj = builtin('subsasgn',obj,s,varargin{:});
        end
    end
end
```

Subscripted assignment to an uninitialized variable now avoids the previous error.

```
clear B
B(1) = MyClass;
```

```
B =

  MyClass with no properties.
```

See Also

Related Examples

- “Subclasses of Built-In Types with Properties” on page 12-53

Overload end for Classes

In this section...

In a standard MATLAB indexing expression, `end` returns the index value of the last element in the dimension in which `end` appears. For example, in `A(4,end)`, the `end` method returns the index of the last element in the second dimension of `A`. You can overload `end` in classes for specialized behavior.

Syntax and Default Behavior

This is the syntax MATLAB uses to call the `end` method.

```
ind = end(A,k,n)
```

- `A` is the object being indexed into.
- `k` is the dimension in the indexing expression where `end` appears.
- `n` is the total number of indices in the expression.
- `ind` is the index value to use in the expression.

Note You cannot call the `end` method directly using this syntax. MATLAB automatically calls the method when it encounters end in an indexing expression.

For example, `A` is a 2-by-3 array of doubles. When MATLAB encounters the expression `A(end,1)`, it calls the `end` method with these arguments.

```
end(A,1,2)
```

- `A` is the object.
- `k = 1` because `end` appears in the first dimension of the indexing expression.
- `n = 2` because the expression has two indices.

The `end` method returns `2`, which is the index of the last element in the first dimension of `A`.

How RedefinesParen Overloads end

Any overload of the `end` method must have the calling syntax `ind = end(A,k,n)`. For example, the modular indexing class `matlab.mixin.indexing.RedefinesParen` has a built-in overload of `end`.

```
function ind = end(obj,k,n)
    sz = size(obj);
    if k < n
        ind = sz(k);
    else
        ind = prod(sz(k:end));
    end
end
```

The `if-else` statement calculates the return value based on where the `end` appears in the indexing expression and whether the indexing expression has values for all of the dimensions of the object array. For example, when `B` is a 2-by-3-by-2 object array of a type that inherits from `RedefinesParen`:

- `k < n`: When `end` is not the last value in the indexing expression, the overload returns the last value in that dimension. For `B(1,end,4)`, `end` returns the size of the second dimension, `3`.
- `k = n`: When `end` is the last element in the indexing expression, the overload handles two cases:
  - If the indexing expression references all the indices, then `prod(sz(k:end))` gives the same result as `sz(k)`. For example, in `B(1,2,end)`, `end` returns `2`.
  - If the indexing expression does not reference all the indices, then `prod(sz(k:end))` returns the product of the size of dimension `k` and the sizes of all unreferenced dimensions. For example, in `B(1,end)`, `end` returns the product of the sizes of the second and third dimensions, `6`.

`RedefinesParen` defines `size` as an abstract method for the class author to implement, so the two methods are dependent on one another for the final behavior. See the “Customize Parentheses Indexing” example for a class that implements a `size` method that provides the expected `end` behavior with an array.

See Also

Related Examples

- “Objects in Index Expressions” on page 17-17

Objects in Index Expressions

In this section...

Objects as Indexes

MATLAB can use objects as indices in indexed expressions. The rules of array indexing apply — indices must be positive integers. Therefore, MATLAB must be able to derive a value from the object that is a positive integer for use in the indexed expression.

Indexed expressions like `X(A)`, where `A` is an object, cause MATLAB to call the `subsindex` function. However, if an indexing expression results in a call to an overloaded method from `matlab.mixin.indexing.RedefinesParen`, `matlab.mixin.indexing.RedefinesDot`, or `matlab.mixin.indexing.RedefinesBrace` defined by class `X`, then MATLAB does not call `subsindex`.

Ways to Implement Objects as Indices

There are several ways to implement indexing of one object by another object, `X(A)`:

- Define a `subsindex` method in the class of `A` that converts `A` to an integer. MATLAB calls `A`'s `subsindex` method to perform indexing operations when the class of `X` does not overload methods from `matlab.mixin.indexing.RedefinesParen`, `matlab.mixin.indexing.RedefinesDot`, or `matlab.mixin.indexing.RedefinesBrace`.
- If the class of `X` overloads methods from `RedefinesParen`, `RedefinesDot`, or `RedefinesBrace` these methods can call the `subsindex` method of `A` explicitly. The class of `A` must implement a `subsindex` method that returns an appropriate value.

subsindex Implementation

`subsindex` must return the value of the object as a zero-based integer index value in the range `0` to `prod(size(X))-1`.

Suppose that you want to use object `A` to index into object `B`. `B` can be a single object or an array, depending on the class designs.

```
C = B(A);
```

Here are two examples of `subsindex` methods. The first assumes you can convert class `A` to a `uint8`. The second assumes class `A` stores an index value in a property.

- The `subsindex` method implemented by class `A` can convert the object to numeric format to be used as an index:

```
function ind = subsindex(obj)
   ind = uint8(obj);
end
```

The class of `obj` implements a `uint8` method to provide the conversion from the object to an integer value.

- Class `A` implements `subsindex` to return a numeric value that is stored in a property:

```
function ind = subsindex(obj)
   ind = obj.ElementIndex;
end
```

Note `subsindex` values are 0-based, not 1-based.

See Also

`matlab.mixin.indexing.RedefinesParen` | `matlab.mixin.indexing.RedefinesDot` | `matlab.mixin.indexing.RedefinesBrace`

Related Examples

- “Overload end for Classes” on page 17-15

Operator Overloading

In this section...

Why Overload Operators

By implementing operators that are appropriate for your class, you can integrate objects of your class into the MATLAB language. For example, objects that contain numeric data can define arithmetic operations like +, *, - so that you can use these objects in arithmetic expressions. By implementing relational operators, you can use objects in conditional statements, like `switch` and `if` statements.

How to Define Operators

You can implement MATLAB operators to work with objects of your class. To implement operators, define the associated class methods.

Each operator has an associated function (e.g., the + operator has an associated `plus.m` function). You can implement any operator by creating a class method with the appropriate name. This method can perform whatever steps are appropriate for the operation being implemented.

For a list of operators and associated function names, see “MATLAB Operators and Associated Functions” on page 17-21.

Object Precedence in Operations

User-defined classes have a higher precedence than built-in classes. For example, suppose `q` is an object of class `double` and `p` is a user-defined class. Both of these expressions generate a call to the `plus` method in the user-define class, if it exists:

```
q + p
p + q
```

Whether this method can add objects of class `double` and the user-defined class depends on how you implement the method.

When `p` and `q` are objects of different classes, MATLAB applies the rules of precedence to determine which method to use.

For more information on how MATLAB determines which method to call, see “Method Invocation” on page 9-11.

Operator Precedence

Overloaded operators retain the original MATLAB precedence for the operator. For information on operator precedence, see “Operator Precedence”.

Sample Implementation — Addable Objects

The `Adder` class implements addition for objects of this class by defining a `plus` method. `Adder` defines addition of objects as the addition of the `NumericData` property values. The `plus` method constructs and returns an `Adder` object whose `NumericData` property value is the result of the addition.

The `Adder` class also implements the less than operator (`<`) by defining a `lt` method. The `lt` method returns a logical value after comparing the values in each object `NumericData` property.

```
classdef Adder
   properties
      NumericData
   end
   methods
      function obj = Adder(val)
         obj.NumericData = val;
      end
      function r = plus(obj1,obj2)
         a = double(obj1);
         b = double(obj2);
         r = Adder(a + b);
      end
      function d = double(obj)
         d = obj.NumericData;
      end
      function tf = lt(obj1,obj2)
         if obj1.NumericData < obj2.NumericData
            tf = true;
         else
            tf = false;
         end
      end
   end
end
```

Using a double converter enables you to add numeric values to `Adder` objects and to perform addition on objects of the class.

```
a = Adder(1:10)
```

```
a =

  Adder with properties:

    NumericData: [1 2 3 4 5 6 7 8 9 10]
```

Add two objects:

```
a + a
```

```
ans =

  Adder with properties:

    NumericData: [2 4 6 8 10 12 14 16 18 20]
```

Add an object with any value that can be cast to double:

```
b = uint8(255) + a

b =

  Adder with properties:

    NumericData: [256 257 258 259 260 261 262 263 264 265]
```

Compare objects `a` and `b` using the `<` operator:

```
a < b

ans =

     1
```

Ensure that your class provides any error checking required to implement your class design.

MATLAB Operators and Associated Functions

The following table lists the function names for MATLAB operators. Implementing operators to work with arrays (scalar expansion, vectorized arithmetic operations, and so on), can also require modifying indexing and concatenation. Use the links in this table to find specific information on each function.

Operation	Method to Define	Description
`a + b`	`plus(a,b)`	Binary addition
`a - b`	`minus(a,b)`	Binary subtraction
`-a`	`uminus(a)`	Unary minus
`+a`	`uplus(a)`	Unary plus
`a.*b`	`times(a,b)`	Element-wise multiplication
`a*b`	`mtimes(a,b)`	Matrix multiplication
`a./b`	`rdivide(a,b)`	Right element-wise division
`a.\b`	`ldivide(a,b)`	Left element-wise division
`a/b`	`mrdivide(a,b)`	Matrix right division
`a\b`	`mldivide(a,b)`	Matrix left division
`a.^b`	`power(a,b)`	Element-wise power
`a^b`	`mpower(a,b)`	Matrix power
`a < b`	`lt(a,b)`	Less than
`a > b`	`gt(a,b)`	Greater than
`a <= b`	`le(a,b)`	Less than or equal to
`a >= b`	`ge(a,b)`	Greater than or equal to
`a ~= b`	`ne(a,b)`	Not equal to
`a == b`	`eq(a,b)`	Equality
`a & b`	`and(a,b)`	Logical AND
`a | b`	`or(a,b)`	Logical OR

Operation	Method to Define	Description
`~a`	`not(a)`	Logical NOT
`a:d:b`	`colon(a,d,b)`	Colon operator
`a:b`	`colon(a,b)`	
`a'`	`ctranspose(a)`	Complex conjugate transpose
`a.'`	`transpose(a)`	Matrix transpose
`[a b]`	`horzcat(a,b,...)`	Horizontal concatenation
`[a; b]`	`vertcat(a,b,...)`	Vertical concatenation
`a(s1,s2,...sn)`	`subsref(a,s)`	Subscripted reference
`a(s1,...,sn) = b`	`subsasgn(a,s,b)`	Subscripted assignment
`b(a)`	`subsindex(a)`	Subscript index

See Also

Related Examples

- “Define Arithmetic Operators” on page 19-12
- “Methods That Modify Default Behavior” on page 17-2

Customize Parentheses Indexing for Mapping Class

This example shows how to customize parentheses indexing for a class. The `MyMap` class stores strings ("keys") that are associated with elements ("values") of a cell array. The class inherits from `matlab.mixin.indexing.RedefinesParen` to define custom behavior for indexing with parentheses. The class supports three customized indexing operations:

- *`instanceName`*`("`*`keyName`*`")` returns the value associated with the key.
- *`instanceName`*`("`*`keyName`*`") = `*`value` adds a new key and associated value.
- *`instanceName`*`("`*`keyName`*`") = []` deletes the key and its associated value.

The full code for the class and its helper function, `validateKeys`, is available at the end of the example.

MyMap Class	Explanation
`classdef MyMap...` `  < matlab.mixin.indexing.RedefinesParen`	Define `MyMap` as a subclass of `RedefinesParen`. and implement its abstract methods

`properties (Access = private)`
`   Keys (:,1) string`
`   Values (:,1) cell`
`end`
	The private properties `Keys` and `Values` are *n*-by-1 vectors.

`methods (Static, Access = public)`
`   function obj = empty(varargin)`
`      if nargin == 0`
`         obj = MyMap(string.empty(0,1),cell.empty(0,1));`
`         return;`
`      end`
 	
`      keys = string.empty(varargin{:});`
`      if ~all(size(keys) == [0, 1])`
`         error("MyMap:MustBeEmptyColumnVector",...`
`            "The only supported empty size is 0x1.");`
`      end`
 	
`      obj = MyMap(keys,cell.empty(varargin{:}));`
`   end`
`end`
	Implementation of the static, abstract method `empty`, which creates a `MyMap` object with no keys or values.

MyMap Class	Explanation

`methods (Access = public)`
`  function obj = MyMap(keys_in,values_in)`
`  if nargin == 0`
`  obj = MyMap.empty(0,1);`
`  return;`
`  end`
 	
`  narginchk(2,2);`
 	
`  if ~all(size(keys_in) == size(values_in))`
`  error("MyMap:InputSizesDoNotMatch",...`
`  "The sizes of the input keys and values must match.");`
`  end`
 	
`  obj.Keys = keys_in;`
`  obj.Values = values_in;`
`  end`
`end`
	The constructor accepts the keys and values as input arguments and ensures the arrays are the same size.

`  function keys = getKeys(obj)`
`  keys = obj.Keys;`
`  end`
 	
`  function values = getValues(obj)`
`  values = obj.Values;`
`  end`
	Two public methods provide read access to the keys and values.

`  function varargout = size(obj,varargin)`
`  [varargout{1:nargout}] = size(obj.Keys,varargin{:});`
`  end`
 	
`  function C = cat(dim,varargin)`
`  error("MyMap:ConcatenationNotSupported",...`
`  "Concatenation is not supported.");`
`  end`
 	
`  function lastKey = end(~,~,~)`
`  error("MyMap:EndNotSupported",...`
`  "Using end with MyMap objects is not supported.");"`
`  end`
	Implementation of the abstract methods `size`, `cat`, and `end`. In this example, `cat` and `end` are not supported.

`methods (Access = private)`
`  function [keyExists,idx] = convertKeyToIndex(obj,keyCellArray)`
`  arguments`
`  obj`
`  keyCellArray cell {validateKeys}`
`  end`
 	
`  requestedKey = keyCellArray{1};`
`  idx = find(contains(obj.Keys,requestedKey));`
`  keyExists = ~isempty(idx);`
`  end`
`end`
	The `parenReference`, `parenAssign`, `parenDelete`, and `parenListLength` methods use the `convertKeyToIndex` helper method. `convertKeyToIndex` uses `validateKeys` to ensure that `keyCellArray` contains only one key. (See the code for `validateKeys` after the end of the class.) `convertKeyToIndex` returns a logical that indicates whether or not the input key exists and, if it does, the index of the key.

MyMap Class	Explanation

`methods (Access = protected)`
`  function varargout = parenReference(obj,indexOp)`
`  [keyExists,idx] = convertKeyToIndex(obj,indexOp(1).Indices);`
 	
`  if ~keyExists`
`  error("MyMap:KeyDoesNotExist",...`
`  "The requested key does not exist.");`
`  end`
 	
`  if numel(indexOp) == 1`
`  nargoutchk(0,1);`
`  varargout{1} = obj.Values{idx};`
`  else`
`  % Additional operations`
`  [varargout{1:nargout}] = obj.Values{idx}.(indexOp(2:end));`
`  end`
`  end`
	Implementation of the abstract method `parenReference`, which handles reference indexing expressions of the form *`instanceName`*`("`*`keyName`*`")`. The method takes an `IndexingOperation` instance as input. That instance, `indexOp`, includes the indexing type (in this case, parentheses) and the index value from the expression being interpreted. `parenReference` passes the index in `indexOp` to `convertKeyToIndex` to verify that the key exists and, if so, return the index of that key `idx`. The method then returns the value that corresponds to the key. If there is more than one indexing operation in `indexOp`, the line labeled "Additional operations" forwards the handling of those operations to MATLAB.

`  function obj = parenAssign(obj,indexOp,varargin)`
`  indicesCell = indexOp(1).Indices;`
`  [keyExists,idx] = convertKeyToIndex(obj,indicesCell);`
 	
`  if numel(indexOp) == 1`
`  value = varargin{1};`
`  if keyExists`
`  obj.Values{idx} = value;`
`  else`
`  obj.Keys(end+1) = indicesCell{1};`
`  obj.Values{end+1} = value;`
`  end`
`  return;`
`  end`
 	
`  if ~keyExists`
`  error("MyMap:MultiLevelAssignKeyDoesNotExist", ...`
`  "Assignment failed because key %s does not exist",...`
`  indicesCell{1});`
`  end`
 	
`  [obj.Values{idx}.(indexOp(2:end))] = varargin{:};`
`  end`
	Implementation of the abstract method `parenAssign`, which handles assignment indexing expressions of the form *`instanceName`*`("`*`keyName`*`")= `*`value`*. If the key referenced exists, then the method assigns the value from the right-hand side of the expression to that key. If the key does not exist and there is only one level of indexing, then the key and value are added to the list.

`  function obj = parenDelete(obj,indexOp)`
`  [keyExists,idx] = convertKeyToIndex(obj,indexOp(1).Indices);`
`  if keyExists`
`  obj.Keys(idx) = [];`
`  obj.Values(idx) = [];`
`  else`
`  error("MyMap:DeleteNonExistentKey",...`
`  "Unable to perform deletion. The key %s does not exist.",...`
`  indexOp(1).Indices{1});`
`  end`
`  end`
	Implementation of the abstract method `parenDelete`, which handles deletion indexing expressions of the form *`instanceName`*`("`*`keyName`*`") = []`. If the key referenced exists, then the method deletes the key and its associated value. If the key does not exist, then the method issues an error.

MyMap Class	Explanation

`function n = parenListLength(obj,indexOp,indexingContext)`
`  [keyExists,idx] = convertKeyToIndex(obj,indexOp(1).Indices);`
`  if ~keyExists`
`  if indexingContext == matlab.indexing.IndexingContext.Assignment`
`  error("MyMap:MultiLevelAssignKeyDoesNotExist",...`
`  "Unable to perform assignment. Key %s does not exist",...`
`  indexOp(1).Indices{1});`
`  end`
`  error("MyMap:KeyDoesNotExist",...`
`  "The requested key does not exist.");`
`  end`
`  n = listLength(obj.Values{idx},indexOp(2:end),indexingContext);`
`  end`
`end`
	Implementation of the abstract method `parenListLength`, which determines the number of values to return from parentheses indexing expressions. The method takes an instance of `matlab.indexing.IndexingContext` as input to determine whether the reference is used in a statement, as a list of arguments to a function, or in an assignment operation.

Expand for Class and Helper Function Code

```
classdef MyMap...
  < matlab.mixin.indexing.RedefinesParen

    properties (Access = private)
        Keys (:,1) string
        Values (:,1) cell
    end

    methods (Static, Access = public)
        function obj = empty(varargin)
            if nargin == 0
                obj = MyMap(string.empty(0,1),cell.empty(0,1));
                return;
            end

            keys = string.empty(varargin{:});
            if ~all(size(keys) == [0, 1])
                error("MyMap:MustBeEmptyColumnVector",...
                    "The only supported empty size is 0x1.");
            end

            obj = MyMap(keys,cell.empty(varargin{:}));
        end
    end

    methods (Access = public)
        function obj = MyMap(keys_in,values_in)
            if nargin == 0
                obj = MyMap.empty(0,1);
                return;
            end

            narginchk(2,2);

            if ~all(size(keys_in) == size(values_in))
                error("MyMap:InputSizesDoNotMatch",...
                    "The sizes of the input keys and values must match.");
            end

            obj.Keys = keys_in;
            obj.Values = values_in;
        end

        function keys = getKeys(obj)
            keys = obj.Keys;
        end

        function values = getValues(obj)
            values = obj.Values;
```

```
        end

        function varargout = size(obj,varargin)
            [varargout{1:nargout}] = size(obj.Keys,varargin{:});
        end

        function C = cat(dim,varargin)
            error("MyMap:ConcatenationNotSupported",...
                "Concatenation is not supported.");
        end

        function lastKey = end(~,~,~)
            error("MyMap:EndNotSupported",...
                "Using end with MyMap objects is not supported.");
        end
    end

    methods (Access = private)
        function [keyExists,idx] = convertKeyToIndex(obj,keyCellArray)
            arguments
                obj
                keyCellArray cell {validateKeys}
            end

            requestedKey = keyCellArray{1};
            idx = find(contains(obj.Keys,requestedKey));
            keyExists = ~isempty(idx);
        end
    end

    methods (Access = protected)
        function varargout = parenReference(obj,indexOp)
            [keyExists,idx] = convertKeyToIndex(obj,indexOp(1).Indices);

            if ~keyExists
                error("MyMap:KeyDoesNotExist",...
                    "The requested key does not exist.");
            end

            if numel(indexOp) == 1
                nargoutchk(0,1);
                varargout{1} = obj.Values{idx};
            else
                [varargout{1:nargout}] = obj.Values{idx}.(indexOp(2:end));
            end
        end

        function obj = parenAssign(obj,indexOp,varargin)
            indicesCell = indexOp(1).Indices;
            [keyExists,idx] = convertKeyToIndex(obj,indicesCell);

            if numel(indexOp) == 1
                value = varargin{1};
                if keyExists
                    obj.Values{idx} = value;
                else
                    obj.Keys(end+1) = indicesCell{1};
                    obj.Values{end+1} = value;
                end
                return;
            end

            if ~keyExists
                error("MyMap:MultiLevelAssignKeyDoesNotExist", ...
                    "Assignment failed because key %s does not exist",...
                indicesCell{1});
            end

            [obj.Values{idx}.(indexOp(2:end))] = varargin{:};
        end

        function obj = parenDelete(obj,indexOp)
```

```
            [keyExists,idx] = convertKeyToIndex(obj,indexOp(1).Indices);
            if keyExists
                obj.Keys(idx) = [];
                obj.Values(idx) = [];
            else
                error("MyMap:DeleteNonExistentKey",...
                    "Unable to perform deletion. The key %s does not exist.",...
                indexOp(1).Indices{1});
            end
        end

        function n = parenListLength(obj,indexOp,indexingContext)
            [keyExists,idx] = convertKeyToIndex(obj,indexOp(1).Indices);
            if ~keyExists
                if indexingContext == matlab.indexing.IndexingContext.Assignment
                    error("MyMap:MultiLevelAssignKeyDoesNotExist", ...
                        "Unable to perform assignment. Key %s does not exist",...
                    indexOp(1).Indices{1});
                end
                error("MyMap:KeyDoesNotExist",...
                    "The requested key does not exist.");
            end
            n = listLength(obj.Values{idx},indexOp(2:end),indexingContext);
         end
    end
end

function validateKeys(requestedKeysCell)
   if numel(requestedKeysCell) == 0
      error("MyMap:MustProvideAtLeastOneKey",...
        "At least one key must be provided.");
   end

   if numel(requestedKeysCell) > 1
      error("MyMap:MustUseOnlyOneSubscript",...
        "Indexing with more than one subscript not supported.");
   end

   requestedKeys = requestedKeysCell{1};

   if ~isstring(requestedKeys)
      error("MyMap:InvalidKey","Keys must be strings.");
   end

   if numel(requestedKeys) == 0
      error("MyMap:MustProvideAtLeastOneKey",...
          "At least one key must be provided.");
   end

   if numel(requestedKeys) > 1
      error("MyMap:UnableToUseMoreThanOneKey",...
          "Unable to use more than one key.");
   end
end
```

Save the code for `MyMap` and `validateKeys` in your MATLAB path. Create an instance of `MyMap` with an initial list of three keys and values.

```
map = MyMap(["apple","cherry","orange"],{1,3,15});
```

Use the `map("keyName")` syntax to return the value corresponding to a specific key.

```
map("cherry")
```

```
ans =

     3
```

Use the `map("keyName") = value` to add a new key to the array.

```
map("banana") = 2;
map("banana")
```

```
ans =

     2
```

Use the `map("keyName") = []` to delete a key and its associated value from the array. Confirm the key is no longer in the array.

```
map("orange") = [];
map("orange")
```

```
Error using MyMap/parenReference (line 88)
The requested key does not exist.
```

See Also

`matlab.mixin.indexing.RedefinesParen` | `matlab.indexing.IndexingOperation`

Forward Indexing Operations

The three mixin classes that enable customizing indexing operations—`matlab.mixin.indexing.RedefinesParen`, `matlab.mixin.indexing.RedefinesDot`, and `matlab.mixin.indexing.RedefinesBrace`—work independently. You can implement all three indexing operations, but you also have the option of just implementing one or two of them. You can also choose to customize just one or two levels of indexing and forward additional operations to another MATLAB object.

For example, this expression shows three levels of indexing:

```
obj(1).prop{7}
```

You can choose to customize only the parentheses indexing by inheriting from `RedefinesParen` and then forward the remaining indexing operations (dot and brace) to the default behaviors. This class fragment inherits from `RedefinesParen`:

```
classdef MyClass < matlab.mixin.indexing.RedefinesParen

   properties (Access = private)
      prop
   end

   methods (Access = protected)
      function varargout = parenReference(A,indexOp)
         idx = indexop(1).Indices;

         % Handle customized parentheses indexing
         temp = A.prop(idx);

         % Foward remaining indexing to temp
         [varargout{1:nargout}] = temp.(indexOp(2:end));
      end
   end
end
```

The forwarding syntax is the dynamic dot indexing syntax with the `IndexingOperation` instance:

```
temp.(indexOp(2:end))
```

This expression handles all of the indexing operations after the first parentheses. In other words, the indexing expression described by `indexOp(2:end)` is forwarded to `temp`. In this example, `indexOp(2)` and `indexOp(3)` are the dot and brace indexing operations, respectively. `temp.(indexOp(2:end))` translates to:

```
temp.prop{7}
```

For customized dot indexing, the forwarding syntax maintains the access permissions from the original context of the main indexing expression. For example, after the parentheses indexing `obj(1).prop{7}` is handled, MATLAB handles the dot indexing, `temp.prop{7}`, using the same context that the initial indexing expression started with. When you call `obj(1).prop{7}` inside the class, the private property `prop` is accessible. When you call `obj(1).prop{7}` outside of the class, `prop` is not accessible.

See Also

`matlab.mixin.indexing.RedefinesParen` | `matlab.mixin.indexing.RedefinesDot` | `matlab.mixin.indexing.RedefinesBrace` | `matlab.indexing.IndexingOperation`

Related Examples

- “Customize Parentheses Indexing for Mapping Class” on page 17-23

Customizing Object Display

Custom Display Interface

In this section...

Command Window Display

MATLAB displays information in the command window when a statement that is not terminated with a semicolon returns a variable. For example, this statement creates a structure with a field that contains the number 7.

```
a.field1 = 7
```

MATLAB displays the variable name, class, and the value.

```
a =

  struct with fields:

    field1: 7
```

MATLAB provides user-defined classes with similar display functionality. User-defined classes can customize how MATLAB displays objects of the class using the API provided by the `matlab.mixin.CustomDisplay` class. To use this API, derive your class from `matlab.mixin.CustomDisplay`.

Default Object Display

MATLAB adds default methods named `disp` and `display` to all MATLAB classes that do not implement their own methods with those names. These methods are not visible, but create the default simple display.

The default simple display consists of the following parts:

- A header showing the class name, and the dimensions for nonscalar arrays.
- A list of all nonhidden public properties, shown in the order of definition in the class.

The actual display depends on whether the object is scalar or nonscalar. Also, there are special displays for a scalar handle to a deleted object and empty object arrays. Objects in all of these states are displayed differently if the objects have no properties.

The `details` function creates the default detailed display. The detailed display adds these items to the simple display:

- Use of fully qualified class names
- Link to `handle` class, if the object is a handle
- Links to `methods`, `events`, and `superclasses` functions executed on the object.

See “Class with Default Object Display” on page 18-11 for an example of how MATLAB displays objects.

Properties Displayed by Default

MATLAB displays object properties that have public get access and are not hidden (see “Property Attributes” on page 8-8). Inherited abstract properties are excluded from display. When the object being displayed is scalar, any dynamic properties attached to the object are also included.

CustomDisplay Class

The `matlab.mixin.CustomDisplay` class provides an interface that you can use to customize object display for your class. To use this interface, derive your class from `CustomDisplay`:

```
classdef MyClass < matlab.mixin.CustomDisplay
```

The `CustomDisplay` class is `HandleCompatible`, so you can use it in combination with both value and handle superclasses.

Note You cannot use `matlab.mixin.CustomDisplay` to derive a custom display for enumeration classes.

---

disp, display, and details

The `CustomDisplay` interface does not allow you to override `disp`, `display`, and `details`. Instead, override any combination of the customization methods defined for this purpose.

Methods for Customizing Object Display

There are two groups of methods that you use to customize object display for your class:

- Part builder methods build the strings used for the standard display. Override any of these methods to change the respective parts of the display.
- State handler methods are called for objects in specific states, like scalar, nonscalar, and so on. Override any of these methods to handle objects in a specific state.

All of these methods have protected access and must be defined as protected in your subclass of `CustomDisplay` (that is, `Access = protected`).

Parts of an Object Display

There are three parts that makeup the standard object display — header, property list, and footer

For example, here is the standard object display for a `containers.Map` object:

```
>> map1 = containers.Map({'Apr','Jul','Nov'},[4,7,11])

map1 =

  Map with properties:

       Count: 3
     KeyType: char
   ValueType: double
```

Header

Property List

The default object display does not include a footer. The detailed display provides more information:

```
>> details(map1)

  3x1 containers.Map handle array with properties:

       Count: 3
     KeyType: 'char'
   ValueType: 'double'

  Methods, Events, Superclasses
```

Footer

You can customize how MATLAB displays objects as a result of expressions that display objects in the command window such as unterminated statements that return objects or calls to `disp` and `display`. The results displayed when calling `details` on an object or object array are not changed by the `CustomDisplay` API.

Part Builder Methods

Each part of the object display has an associated method that assembles the respective part of the display.

Method	Purpose	Default
`getHeader`	Create the text used for the header.	Returns the `char` vectors, `[class(obj), ' with properties:']` linking the class name to a help popup
`getPropertyGroups`	Define how and what properties display, including order, values, and grouping.	Returns an array of `PropertyGroup` objects, which determines how to display the properties
`getFooter`	Create the text used for the footer.	There are two footers: • Simple display — Returns an empty `char` vector • Detailed display — Returns linked calls to `methods`, `events`, and `superclasses` for this class

Object States That Affect Display

There are four object states that affect how MATLAB displays objects:

- Valid scalar object
- Nonscalar object array
- Empty object array
- Scalar handle to a deleted object

State Handler Methods

Each object state has an associated method that MATLAB calls whenever displaying objects that are in that particular state.

State Handler Method	Called for Object in This State
`displayScalarObject`	`(isa(obj,'handle') && isvalid(obj)) && prod(size(obj)) == 1`
`displayNonScalarObject`	`prod(size(obj)) > 1`
`displayEmptyObject`	`prod(size(obj)) == 0`
`displayScalarHandleToDeletedObject`	`isa(obj,'handle') && isscalar(obj) && ~isvalid(obj)`

Utility Methods

The `CustomDisplay` class provides utility methods that return strings that are used in various parts of the different display options. These static methods return text that simplifies the creation of customized object displays.

If the computer display does not support hypertext linking, the strings are returned without the links.

Method	Inputs	Outputs
`convertDimensionsToString`	Valid object array	Object dimensions converted to a `char` vector; determined by calling `size(obj)`
`displayPropertyGroups`	`PropertyGroup` array	Displays the titles and property groups defined
`getClassNameForHeader`	Object	Simple class name linked to the object's documentation
`getDeletedHandleText`	None	Text `'handle to deleted'` linked to the documentation on deleted handles
`getDetailedFooter`	Object	Text containing phrase `'Methods, Events, Superclasses'`, with each link executing the respective command on the input object
`getDetailedHeader`	Object	Text containing linked class name, link to handle page (if handle class) and `'with properties:'`

Method	Inputs	Outputs
`getHandleText`	None	Text `'handle'` linked to a section of the documentation that describes handle objects
`getSimpleHeader`	Object	Text containing linked class name and the phrase `'with properties:'`

See Also

Related Examples

- “How CustomDisplay Works” on page 18-7

How CustomDisplay Works

In this section...

Steps to Display an Object

When displaying an object, MATLAB determines the state of the object and calls the appropriate method for that state (see “Object States That Affect Display” on page 18-5).

For example, suppose `obj` is a valid scalar object of a class derived from `CustomDisplay`. If you type `obj` at the command line without terminating the statement with a semicolon:

```
>> obj
```

The following sequence results in the display of `obj`:

1. MATLAB determines the class of `obj` and calls the `disp` method to display the object.
2. `disp` calls `size` to determine if `obj` is scalar or nonscalar
3. When `obj` is a scalar handle object, `disp` calls `isvalid` to determine if `obj` is the handle of a deleted object. Deleted handles in nonscalar arrays do not affect the display.
4. `disp` calls the state handler method for an object of the state of `obj`. In this case, `obj` is a valid scalar that results in a call to:

   ```
   displayScalarObject(obj)
   ```
5. `displayScalarObject` calls the display part-builder methods to provide the respective header, property list, and footer.

   ```
   ...
   header = getHeader(obj);
   disp(header)
   ...
   groups = getPropertyGroups(obj)
   displayPropertyGroups(obj,groups)
   ...
   footer = getFooter
   disp(footer)
   ```

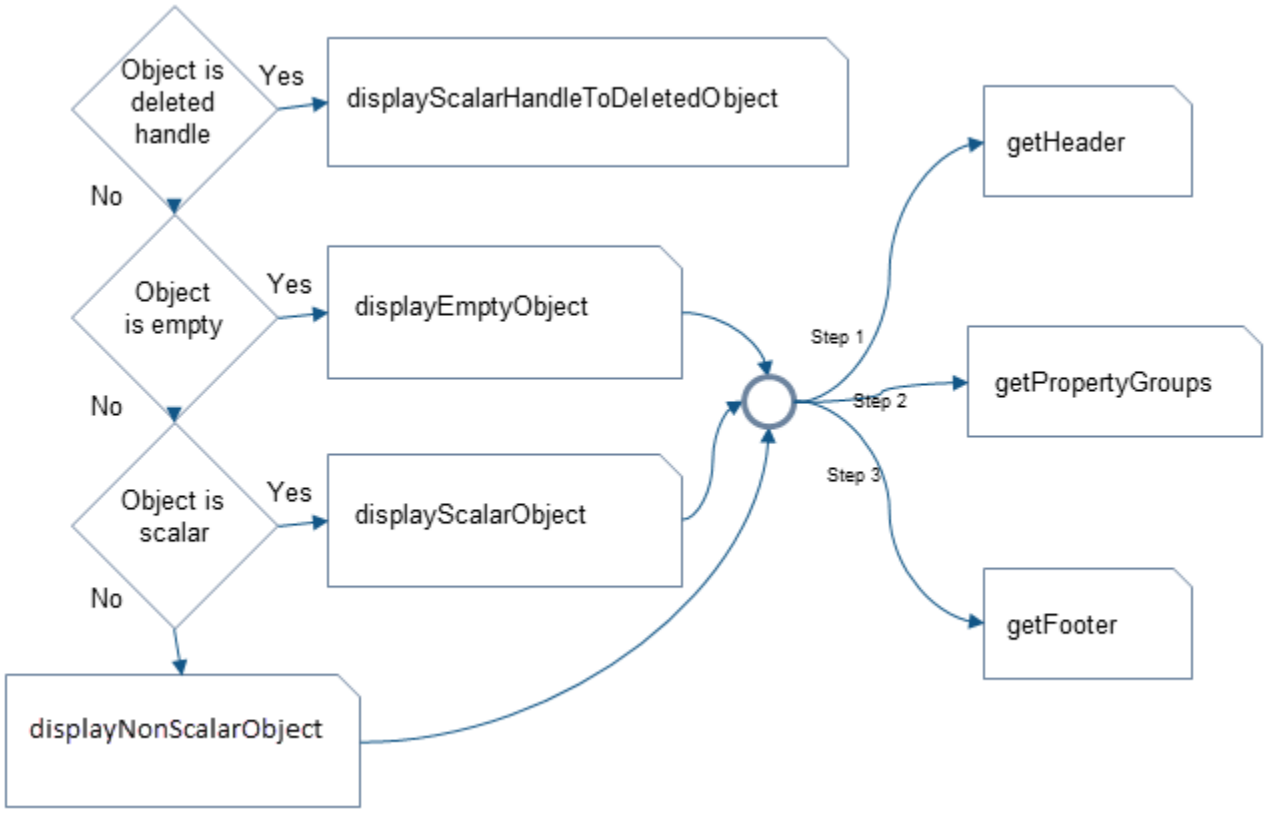
MATLAB follows a similar sequence for nonscalar object arrays and empty object arrays.

In the case of scalar handles to deleted objects, `disp` calls the `displayScalarHandleToDeletedObject` method, which displays the default text for handles to deleted objects without calling any part-builder methods.

Methods Called for a Given Object State

The following diagram illustrates the methods called to display an object that derives from `CustomDisplay`. The `disp` method calls the state handler method that is appropriate for the state of the object or object array being displayed.

Only an instance of a handle class can be in a state of scalar handle to a deleted object.



See Also

Related Examples

- “Class with Default Object Display” on page 18-11

Role of size Function in Custom Displays

In this section...

How size Is Used

In the process of building the custom display, `CustomDisplay` methods call the `size` function at several points:

- `disp` calls `size` to determine which state handler method to invoke.
- The default `getHeader` method calls `size` to determine whether to display a scalar or nonscalar header.
- The default `displayPropertyGroups` method calls `size` to determine if it should look up property values when the property group is a cell array of property names. By default, only scalar objects display the values of properties.

Precautions When Overloading size

If your class overloads the `size` function, then MATLAB calls the overloading version. You must ensure that the implementation of `size` is consistent with the way you want to display objects of the class.

An unusual or improper implementation of `size` can result in undesirable display behavior. For example, suppose a class overloads `size` reports an object as scalar when it is not. In this class, a property list consisting of a cell array of strings results in the property values of the first object of the array being displayed. This behavior can give the impression that all objects in the array have the same property values.

However, reporting an object as scalar when in fact the object is empty results in the object displaying as an empty object array. The default methods of the `CustomDisplay` interface always determine if the input is an empty array before attempting to access property values.

As you override `CustomDisplay` methods to implement your custom object display, consider how an overloading size method can affect the result.

See Also

Related Examples

- “Methods That Modify Default Behavior” on page 17-2

Customize Display for Heterogeneous Arrays

You can call only sealed methods on nonscalar heterogeneous arrays. If you want to customize classes that are part of a heterogeneous hierarchy, you must override and declare as `Sealed` all the methods that are part of the `CustomDisplay` interface.

The versions of `disp` and `display` that are inherited from `matlab.mixin.CustomDisplay` are sealed. However, these methods call all of the part builder (“Part Builder Methods” on page 18-4) and state handler methods (“State Handler Methods” on page 18-5).

To use the `CustomDisplay` interface, the root class of the heterogeneous hierarchy can declare these methods as `Sealed` and `Access = protected`.

If you do not need to override a particular method, then call the superclass method, as shown in the following code.

For example, the following code shows modifications to the `getPropertyGroups` and `displayScalarObject` methods, while using the superclass implementation of all others.

```
classdef RootClass < matlab.mixin.CustomDisplay & matlab.mixin.Heterogeneous
   %...
   methods (Sealed, Access = protected)
      function header = getHeader(obj)
         header = getHeader@matlab.mixin.CustomDisplay(obj);
      end

      function groups = getPropertyGroups(obj)
         % Override of this method
         % ...
      end

      function footer = getFooter(obj)
         footer = getFooter@matlab.mixin.CustomDisplay(obj);
      end

      function displayNonScalarObject(obj)
         displayNonScalarObject@matlab.mixin.CustomDisplay(obj);
      end

      function displayScalarObject(obj)
         % Override of this method
         % ...
      end

      function displayEmptyObject(obj)
         displayEmptyObject@matlab.mixin.CustomDisplay(obj);
      end

      function displayScalarHandleToDeletedObject(obj)
         displayScalarHandleToDeletedObject@matlab.mixin.CustomDisplay(obj);
      end
   end
end
```

You do not need to declare the inherited static methods as `Sealed`.

See Also

Related Examples

- “Designing Heterogeneous Class Hierarchies” on page 10-22

Class with Default Object Display

In this section...

The EmployeeInfo Class

The `EmployeeInfo` class defines a number of properties to store information about company employees. This simple class serves as the example class used in display customization sample classes.

`EmployeeInfo` derives from the `matlab.mixin.CustomDisplay` class to enable customization of the object display.

`EmployeeInfo` is also a handle class. Therefore instances of this class can be in the state referred to as a handle to a deleted object. This state does not occur with value classes (classes not derived from `handle`).

```
classdef EmployeeInfo < handle & matlab.mixin.CustomDisplay
   properties
      Name
      JobTitle
      Department
      Salary
      Password
   end
   methods
      function obj = EmployeeInfo
         obj.Name       = input('Name: ');
         obj.JobTitle   = input('Job Title: ');
         obj.Department = input('Department: ');
         obj.Salary     = input('Salary: ');
         obj.Password   = input('Password: ');
      end
   end
end
```

The `matlab.mixin.CustomDisplay` is handle compatible. Therefore, superclasses can be either handle or value classes.

Default Display — Scalar

Here is the creation and display of a scalar `EmployeeInfo` object. By default, MATLAB displays properties and their values for scalar objects.

Provide inputs for the constructor:

```
>>Emp123 = EmployeeInfo;
Name: 'Bill Tork'
Job Title: 'Software Engineer'
Department: 'Product Development'
Salary: 1000
Password: 'bill123'
```

Display the object:

```
>>Emp123

Emp123 =

  EmployeeInfo with properties:

         Name: 'Bill Tork'
     JobTitle: 'Software Engineer'
   Department: 'Product Development'
       Salary: 1000
     Password: 'bill123'
```

Testing for Scalar Objects

To test for scalar objects, use `isscalar`.

Default Display — Nonscalar

The default display for an array of objects does not show property values. For example, concatenating two `EmployeeInfo` objects generates this display:

```
>>[Emp123,Emp124]
ans

  1x2 EmployeeInfo array with properties:

    Name
    JobTitle
    Department
    Salary
    Password
```

Testing for Nonscalar Objects

To test for nonscalar objects, use a negated call to `isscalar` .

Default Display — Empty Object Array

An empty object array has at least one dimension equal to zero.

```
>> Empt = EmployeeInfo.empty(0,5)

Empt =

  0x5 EmployeeInfo array with properties:

    Name
```

```
    JobTitle
    Department
    Salary
    Password
```

Testing for Empty Object Arrays

Use `isempty` to test for empty object arrays. An empty object array is not scalar because its dimensions can never be 1–by-1.

```
>> emt = EmployeeInfo.empty

emt =

  0x0 EmployeeInfo array with properties:

    Name
    JobTitle
    Department
    Salary
    Password

>> isscalar(emt)

ans =

     0
```

Default Display — Handle to Deleted Object

When a handle object is deleted, the handle variable can remain in the workspace.

```
>> delete(Emp123)
>> Emp123
Emp123 =
  handle to deleted EmployeeInfo
```

Testing for Handles to Deleted Objects

To test for a handle to a deleted object, use `isvalid`.

Note `isvalid` is a handle class method. Calling `isvalid` on a value class object causes an error.

Default Display — Detailed Display

The details method does not support customization and always returns the standard detailed display:

```
details(Emp123)
 EmployeeInfo handle with properties:

          Name: 'Bill Tork'
      JobTitle: 'Software Engineer'
    Department: 'Product Development'
        Salary: 1000
      Password: 'bill123'
```

```
Methods, Events, Superclasses
```

See Also

Related Examples

- “Custom Display Interface” on page 18-2

Choose a Technique for Display Customization

In this section...

Ways to Implement a Custom Display

The way you customize object display using the `matlab.mixin.CustomDisplay` class depends on:

- What parts of the display you want to customize
- What object states you want to use the custom display

If you are making small changes to the default layout, then override the relevant part builder methods (“Part Builder Methods” on page 18-4). For example, suppose you want to:

- Change the order or value of properties, display a subset of properties, or create property groups
- Modify the header text
- Add a footer

If you are defining a nonstandard display for a particular object state (scalar, for example), then the best approach is to override the appropriate state handler method (“State Handler Methods” on page 18-5).

In some cases, a combination of method overrides might be the best approach. For example, your implementation of `displayScalarObject` might

- Use some of the utility methods (“Utility Methods” on page 18-5) to build your own display strings using parts from the default display
- Call a part builder method to get the default text for that particular part of the display
- Implement a completely different display for scalar objects.

Once you override any `CustomDisplay` method, MATLAB calls your override in all cases where the superclass method would have been called. For example, if you override the `getHeader` method, your override must handle all cases where a state handler method calls `getHeader`. (See “Methods Called for a Given Object State” on page 18-7)

Sample Approaches Using the Interface

Here are some simple cases that show what methods to use for the particular customized display.

Change the Display of Scalar Objects

Use a nonstandard layout for scalar object display that is fully defined in the `displayScalarObject` method:

```
classdef MyClass < matlab.mixin.CustomDisplay
   ...
   methods (Access = protected)
      function displayScalarObject(obj)
         % Implement the custom display for scalar obj
```

```
      end
   end
end
```

Custom Property List with Standard Layout

Use standard display layout, but create a custom property list for scalar and nonscalar display:

```
classdef MyClass < matlab.mixin.CustomDisplay
   ...
   methods(Access = protected)
      function groups = getPropertyGroups(obj)
         % Return PropertyGroup instances
      end
   end
end
```

Custom Property List for Scalar Only

Use standard display layout, but create a custom property list for scalar only. Call the superclass `getPropertyGroups` for the nonscalar case.

```
classdef MyClass < matlab.mixin.CustomDisplay
   properties
      Prop1
      Prop2
      Prop3
   end
   methods(Access = protected)
      function groups = getPropertyGroups(obj)
         if isscalar(obj)
            % Scalar case: change order
            propList = {'Prop2','Prop1','Prop3'};
            groups = matlab.mixin.util.PropertyGroup(propList)
         else
            % Nonscalar case: call superclass method
            groups = getPropertyGroups@matlab.mixin.CustomDisplay(obj);
         end
      end
   end
end
```

Custom Property List with Modified Values

Change the values displayed for some properties in the scalar case by creating property/value pairs in a `struct`. This `getPropertyGroups` method displays only `Prop1` and `Prop2`, and displays the value of `Prop2` as `Prop1` divided by `Prop3`.

```
classdef MyClass < matlab.mixin.CustomDisplay
   properties
      Prop1
      Prop2
      Prop3
   end
   methods(Access = protected)
      function groups = getPropertyGroups(obj)
         if isscalar(obj)
            % Specify the values to be displayed for properties
            propList = struct('Prop1',obj.Prop1,...
               'Prop2',obj.Prop1/obj.Prop3);
            groups = matlab.mixin.util.PropertyGroup(propList)
         else
            % Nonscalar case: call superclass method
            groups = getPropertyGroups@matlab.mixin.CustomDisplay(obj);
         end
      end
```

```
    end
end
```

Complete Class Definitions

For complete class implementations, see these sections:

- “Customize Property Display” on page 18-18

  “Customize Header, Property List, and Footer” on page 18-21

  “Customize Display of Scalar Objects” on page 18-26

  “Customize Display of Object Arrays” on page 18-30

Customize Property Display

In this section...

Objective

Change the order and number of properties displayed for an object of your class.

Change the Property Order

Suppose your class definition contains the following property definition:

```
properties
       Name
       JobTitle
       Department
       Salary
       Password
end
```

In the default scalar object display, MATLAB displays all the public properties along with their values. However, you want to display only `Department`, `JobTitle`, and `Name`, in that order. You can do this by deriving from `CustomDisplay` and overriding the `getPropertyGroups` method.

Your override

- Defines method `Access` as `protected` to match the definition in the `CustomDisplay` superclass
- Creates a cell array of property names in the desired order
- Returns a `PropertyGroup` object constructed from the property list cell array

```
methods (Access = protected)
   function propgrp = getPropertyGroups(~)
      proplist = {'Department','JobTitle','Name'};
      propgrp = matlab.mixin.util.PropertyGroup(proplist);
   end
end
```

When you create a `PropertyGroup` object using a cell array of property names, MATLAB automatically

- Adds the property values for a scalar object display
- Uses the property names without values for a nonscalar object display (including empty object arrays)

The `getPropertyGroups` method is not called to create the display for a scalar handle to a deleted object.

Change the Values Displayed for Properties

Given the same class properties used in the previous section, you can change the value displayed for properties by building the property list as a `struct` and specifying values for property names. This

override of the `getPropertyGroups` method uses the default property display for nonscalar objects by calling the superclass `getPropertyGroups` method. For scalar objects, the override:

- Changes the value displayed for the `Password` property to a `'*'` character for each character in the password.
- Displays the text `'Not Available'` for the `Salary` property.

```
methods (Access = protected)
   function propgrp = getPropertyGroups(obj)
      if ~isscalar(obj)
         propgrp = getPropertyGroups@matlab.mixin.CustomDisplay(obj);
      else
         pd(1:length(obj.Password)) = '*';
         propList = struct('Department',obj.Department,...
            'JobTitle',obj.JobTitle,...
            'Name',obj.Name,...
            'Salary','Not available',...
            'Password',pd);
         propgrp = matlab.mixin.util.PropertyGroup(propList);
      end
   end
end
```

The object display looks like this:

```
  EmployeeInfo with properties:

    Department: 'Product Development'
      JobTitle: 'Software Engineer'
          Name: 'Bill Tork'
        Salary: 'Not available'
      Password: '*******'
```

Full Class Listing

```
classdef EmployeeInfo < handle & matlab.mixin.CustomDisplay
   properties
      Name
      JobTitle
      Department
      Salary
      Password
   end
   methods
      function obj = EmployeeInfo
         obj.Name       = input('Name: ');
         obj.JobTitle   = input('Job Title: ');
         obj.Department = input('Department: ');
         obj.Salary     = input('Salary: ');
         obj.Password   = input('Password: ');
      end
   end
   methods (Access = protected)
      function propgrp = getPropertyGroups(obj)
         if ~isscalar(obj)
            propgrp = getPropertyGroups@matlab.mixin.CustomDisplay(obj);
         else
            pd(1:length(obj.Password)) = '*';
            propList = struct('Department',obj.Department,...
               'JobTitle',obj.JobTitle,...
               'Name',obj.Name,...
               'Salary','Not available',...
               'Password',pd);
            propgrp = matlab.mixin.util.PropertyGroup(propList);
         end
      end
   end
end
```

See Also

Related Examples

- “Choose a Technique for Display Customization” on page 18-15

Customize Header, Property List, and Footer

In this section...

Objective

Customize each of the three parts of the display — header, property groups, and footer.

Design of Custom Display

Note This example uses the `EmployeeInfo` class described in the “Class with Default Object Display” on page 18-11 section.

For the header:

- Use default header for nonscalar object arrays.
- Build header text with linked class name and department name (from `Department` property)

For properties:

- Nonscalar object arrays display a subset of property names in a different order than the default.
- Scalar objects create two property groups that have titles (`Public Info` and `Personal Info`).

For the footer:

- Add a footer to the display, only when the object is a valid scalar that displays property values.

Here is the customized display of an object of the `EmployeeInfo` class.

```
Emp123 =

EmployeeInfo Dept: Product Development

   Public Info
        Name: 'Bill Tork'
    JobTitle: 'Software Engineer'

   Personal Info
      Salary: 1000
    Password: 'bill123'

Company Private
```

Here is the custom display of an array of `EmployeeInfo` objects:

```
[Emp123,Emp124]

ans =

  1x2 EmployeeInfo array with properties:

    Department
    Name
    JobTitle
```

Here is the display of an empty object array:

```
>> EmployeeInfo.empty(0,5)

ans =

  0x5 EmployeeInfo array with properties:

    Department
    Name
    JobTitle
```

Here is the display of a handle to a deleted object (`EmployeeInfo` is a handle class):

```
>> delete(Emp123)
>> Emp123

Emp123 =

  handle to deleted EmployeeInfo
```

Implementation

The `EmployeeInfo` class overrides three `matlab.mixin.CustomDisplay` methods to implement the display shown:

- `getHeader`
- `getPropertyGroups`
- `getFooter`

Each method must produce the desired results with each of the following inputs:

- Scalar object
- Nonscalar object array
- Empty object array

getHeader Method Override

MATLAB calls `getHeader` to get the header text. The `EmployeeInfo` class overrides this method to implement the custom header for scalar display. Here is how it works:

- Nonscalar (including empty object) arrays call the superclass `getHeader`, which returns the default header.
- Scalar handles to deleted objects do not result in a call to `getHeader`.

- Scalar inputs build a custom header using the `getClassNameForHeader` static method to return linked class name text, and the value of the `Department` property.

Here is the `EmployeeInfo` override of the `getHeader` method. The required protected access is inherited from the superclass.

```
methods (Access = protected)
   function header = getHeader(obj)
      if ~isscalar(obj)
         header = getHeader@matlab.mixin.CustomDisplay(obj);
      else
         className = matlab.mixin.CustomDisplay.getClassNameForHeader(obj);
         newHeader = [className,' Dept: ',obj.Department];
         header = sprintf('%s\n',newHeader);
      end
   end
end
```

getPropertyGroups Override

MATLAB calls `getPropertyGroups` to get the `PropertyGroup` objects, which control how properties are displayed. This method override defines two different property lists depending on the object's state:

- For nonscalar inputs, including empty arrays and arrays containing handles to deleted objects, create a property list as a cell array to reorder properties.

  By default, MATLAB does not display property values for nonscalar inputs.
- For scalar inputs, create two property groups with titles. The scalar code branch lists properties in a different order than the nonscalar case and includes `Salary` and `Password` properties. MATLAB automatically assigns property values.
- Scalar handles to deleted objects do not result in a call to `getPropertyGroups`.

Both branches return a `matlab.mixin.util.PropertyGroup` object, which determines how to displays the object properties.

Here is the `EmployeeInfo` override of the `getPropertyGroups` method. The protected access is inherited from the superclass.

```
methods (Access = protected)
   function propgrp = getPropertyGroups(obj)
      if ~isscalar(obj)
         propList = {'Department','Name','JobTitle'};
         propgrp = matlab.mixin.util.PropertyGroup(propList);
      else
         gTitle1 = 'Public Info';
         gTitle2 = 'Personal Info';
         propList1 = {'Name','JobTitle'};
         propList2 = {'Salary','Password'};
         propgrp(1) = matlab.mixin.util.PropertyGroup(propList1,gTitle1);
         propgrp(2) = matlab.mixin.util.PropertyGroup(propList2,gTitle2);
      end
   end
end
```

getFooter Override

MATLAB calls `getFooter` to get the footer text. The `EmployeeInfo` `getFooter` method defines a footer for the display, which is included only when the input is a valid scalar object. In all other cases, `getFooter` returns an empty `char` vector.

Scalar handles to deleted objects do not result in a call to `getFooter`.

```
methods (Access = protected)
   function footer = getFooter(obj)
      if isscalar(obj)
         footer = sprintf('%s\n','Company Private');
      else
         footer = '';
      end
   end
end
```

Complete Class Listing

```
classdef EmployeeInfo < handle & matlab.mixin.CustomDisplay
   properties
      Name
      JobTitle
      Department
      Salary
      Password
   end
   methods
      function obj = EmployeeInfo
         obj.Name       = input('Name: ');
         obj.JobTitle   = input('Job Title: ');
         obj.Department = input('Department: ');
         obj.Salary     = input('Salary: ');
         obj.Password   = input('Password: ');
      end
   end

   methods (Access = protected)
      function header = getHeader(obj)
         if ~isscalar(obj)
            header = getHeader@matlab.mixin.CustomDisplay(obj);
         else
            className = matlab.mixin.CustomDisplay.getClassNameForHeader(obj);
            newHeader = [className,' Dept: ',obj.Department];
            header = sprintf('%s\n',newHeader);
         end
      end

      function propgrp = getPropertyGroups(obj)
         if ~isscalar(obj)
            propList = {'Department','Name','JobTitle'};
            propgrp = matlab.mixin.util.PropertyGroup(propList);
         else
            gTitle1 = 'Public Info';
            gTitle2 = 'Personal Info';
            propList1 = {'Name','JobTitle'};
            propList2 = {'Salary','Password'};
            propgrp(1) = matlab.mixin.util.PropertyGroup(propList1,gTitle1);
            propgrp(2) = matlab.mixin.util.PropertyGroup(propList2,gTitle2);
         end
      end

      function footer = getFooter(obj)
         if isscalar(obj)
            footer = sprintf('%s\n','Company Private');
         else
            footer = '';
         end
      end
   end
end
```

See Also

Related Examples

- “Choose a Technique for Display Customization” on page 18-15

Customize Display of Scalar Objects

In this section...

Objective

Customize the display of scalar objects.

Design of Custom Display

Note This example uses the `EmployeeInfo` class described in the “Class with Default Object Display” on page 18-11 section.

The objective of this customized display is to:

- Modify the header to include the department name obtained from the `Department` property
- Group properties into two categories titled `Public Info` and `Personal Info`.
- Modify which properties are displayed
- Modify the values displayed for `Personal Info` category
- Use the default displayed for nonscalar objects, including empty arrays, and scalar deleted handles

For example, here is the customized display of an object of the `EmployeeInfo` class.

```
Emp123 =

EmployeeInfo Dept: Product Development

   Public Info
        Name: 'Bill Tork'
    JobTitle: 'Software Engineer'

   Personal Info
      Salary: 'Level: 10'
    Password: '*******'
```

Implementation

The `EmployeeInfo` class overrides two `matlab.mixin.CustomDisplay` methods to implement the display shown:

- `displayScalarObject` — Called to display valid scalar objects
- `getPropertyGroups` — Builds the property groups for display

displayScalarObject Method Override

MATLAB calls `displayScalarObject` to display scalar objects. The `EmployeeInfo` class overrides this method to implement the scalar display. Once overridden, this method must control all aspects of scalar object display, including creating the header, property groups, and footer, if used.

This implementation:

- Builds a custom header using the `getClassNameForHeader` static method to return linked class name text and the value of the `Department` property to get the department name.
- Uses `sprintf` to add a new line to the header text
- Displays the header with the built-in `disp` function.
- Calls the `getPropertyGroups` override to define the property groups (see following section).
- Displays the property groups using the `displayPropertyGroups` static method.

Here is the `EmployeeInfo` override of the `displayScalarObject` method. The required protected access is inherited from the superclass.

```
methods (Access = protected)
   function displayScalarObject(obj)
      className = matlab.mixin.CustomDisplay.getClassNameForHeader(obj);
      scalarHeader = [className,' Dept: ',obj.Department];
      header = sprintf('%s\n',scalarHeader);
      disp(header)
      propgroup = getPropertyGroups(obj);
      matlab.mixin.CustomDisplay.displayPropertyGroups(obj,propgroup)
   end
end
```

getPropertyGroups Override

MATLAB calls `getPropertyGroups` when displaying scalar or nonscalar objects. However, MATLAB does not call this method when displaying a scalar handle to a deleted object.

The `EmployeeInfo` class overrides this method to implement the property groups for scalar object display.

This implementation calls the superclass `getPropertyGroups` method if the input is not scalar. If the input is scalar, this method:

- Defines two titles for the two groups
- Creates a cell array of property names that are included in the first group. MATLAB adds the property values for the display
- Creates a `struct` array of property names with associated property values for the second group. Using a `struct` instead of a cell array enables you to replace the values that are displayed for the `Salary` and `Password` properties without changing the personal information stored in the object properties.
- Constructs two `matlab.mixin.util.PropertyGroup` objects, which are used by the `displayScalarObject` method.

Here is the `EmployeeInfo` override of the `getPropertyGroups` method. The required protected access is inherited from the superclass.

```
methods (Access = protected)
   function propgrp = getPropertyGroups(obj)
```

```
      if ~isscalar(obj)
         propgrp = getPropertyGroups@matlab.mixin.CustomDisplay(obj);
      else
         gTitle1 = 'Public Info';
         gTitle2 = 'Personal Info';
         propList1 = {'Name','JobTitle'};
         pd(1:length(obj.Password)) = '*';
         level = round(obj.Salary/100);
         propList2 = struct('Salary',...
            ['Level: ',num2str(level)],...
            'Password',pd);
         propgrp(1) = matlab.mixin.util.PropertyGroup(propList1,gTitle1);
         propgrp(2) = matlab.mixin.util.PropertyGroup(propList2,gTitle2);
      end
   end
end
```

Complete Class Listing

```
classdef EmployeeInfo4 < handle & matlab.mixin.CustomDisplay
   properties
      Name
      JobTitle
      Department
      Salary
      Password
   end
   methods
      function obj = EmployeeInfo4
         obj.Name       = input('Name: ');
         obj.JobTitle   = input('Job Title: ');
         obj.Department = input('Department: ');
         obj.Salary     = input('Salary: ');
         obj.Password   = input('Password: ');
      end
   end

   methods (Access = protected)
      function displayScalarObject(obj)
         className = matlab.mixin.CustomDisplay.getClassNameForHeader(obj);
         scalarHeader = [className,' Dept: ',obj.Department];
         header = sprintf('%s\n',scalarHeader);
         disp(header)
         propgroup = getPropertyGroups(obj);
         matlab.mixin.CustomDisplay.displayPropertyGroups(obj,propgroup)
      end

      function propgrp = getPropertyGroups(obj)
         if ~isscalar(obj)
            propgrp = getPropertyGroups@matlab.mixin.CustomDisplay(obj);
         else
            % property groups for scalars
            gTitle1 = 'Public Info';
            gTitle2 = 'Personal Info';
            propList1 = {'Name','JobTitle'};
            pd(1:length(obj.Password)) = '*';
            level = round(obj.Salary/100);
            propList2 = struct('Salary',...
               ['Level: ',num2str(level)],...
               'Password',pd);
            propgrp(1) = matlab.mixin.util.PropertyGroup(propList1,gTitle1);
            propgrp(2) = matlab.mixin.util.PropertyGroup(propList2,gTitle2);
         end
      end
   end
end
```

See Also

Related Examples

- “Choose a Technique for Display Customization” on page 18-15

Customize Display of Object Arrays

In this section...

Objective

Customize the display of nonscalar objects, including empty object arrays.

Design of Custom Display

Note This example uses the `EmployeeInfo` class described in the “Class with Default Object Display” on page 18-11 section.

The objective of this customized display is to:

- Construct a custom header using some elements of the default header
- Display a subset of property-specific information for each object in the array.
- List handles to deleted objects in the array using a `char` vector with links to documentation for handle objects and the class.
- Display empty objects with a slight modification to the default header

Here is the customized display of an array of three `EmployeeInfo` objects

```
1x3 EmployeeInfo array with members:
1. Employee:
          Name: 'Bill Tork'
    Department: 'Product Development'

2. Employee:
          Name: 'Alice Blackwell'
    Department: 'QE'

3. Employee:
          Name: 'Nancy Green'
    Department: 'Documentation'
```

Deleted object handles in the array indicate their state:

```
1x3 EmployeeInfo members:

1. Employee:
          Name: 'Bill Tork'
    Department: 'Product Development'

2. handle to deleted EmployeeInfo
```

```
3. Employee:
           Name: 'Nancy Green'
     Department: 'Documentation'
```

To achieve the desired result, the `EmployeeInfo` class overrides the following methods of the `matlab.mixin.CustomDisplay` class:

- `displayNonScalarObject` — Called to display nonempty object arrays
- `displayEmptyObject` — Called to display empty object arrays

The displayNonScalarObject Override

MATLAB calls the `displayNonScalarObject` method to display object arrays. The override of this method in the `EmployeeInfo` class:

- Builds header text using `convertDimensionsToString` to obtain the array size and `getClassNameForHeader` to get the class name with a link to the help for that class.
- Displays the modified header text.
- Loops through the elements in the array, building two different subheaders depending on the individual object state. In the loop, this method:
  - Detects handles to deleted objects (using the `isvalid` handle class method). Uses `getDeletedHandleText` and `getClassNameForHeader` to build text for array elements that are handles to deleted objects.
  - Builds a custom subheader for valid object elements in the array
- Creates a `PropertyGroup` object containing the `Name` and `Department` properties for valid `objects`
- Uses the `displayPropertyGroups` static method to generate the property display for valid objects.

Here is the implementation of `displayNonScalarObjects`:

```
methods (Access = protected)
   function displayNonScalarObject(objAry)
      dimStr = matlab.mixin.CustomDisplay.convertDimensionsToString(objAry);
      cName = matlab.mixin.CustomDisplay.getClassNameForHeader(objAry);
      headerStr = [dimStr,' ',cName,' members:'];
      header = sprintf('%s\n',headerStr);
      disp(header)
      for ix = 1:length(objAry)
         o = objAry(ix);
         if ~isvalid(o)
            str1 = matlab.mixin.CustomDisplay.getDeletedHandleText;
            str2 = matlab.mixin.CustomDisplay.getClassNameForHeader(o);
            headerInv = [str1,' ',str2];
            tmpStr = [num2str(ix),'. ',headerInv];
            numStr = sprintf('%s\n',tmpStr);
            disp(numStr)
         else
            numStr = [num2str(ix),'. Employee:'];
            disp(numStr)
            propList = struct('Name',o.Name,...
               'Department',o.Department);
            propgrp = matlab.mixin.util.PropertyGroup(propList);
            matlab.mixin.CustomDisplay.displayPropertyGroups(o,propgrp);
         end
      end
   end
end
```

The displayEmptyObject Override

MATLAB calls the `displayEmptyObject` method to display empty object arrays. The implementation of this method in the `EmployeeInfo` class builds a custom header for empty objects following these steps:

- Gets the array dimensions in character format using the `convertDimensionsToString` static method.
- Gets text with the class name linked to the `helpPopup` function using the `getClassNameForHeader` static method.
- Builds and displays the custom text for empty arrays.

```
methods (Access = protected)
   function displayEmptyObject(obj)
      dimstr = matlab.mixin.CustomDisplay.convertDimensionsToString(obj);
      className = matlab.mixin.CustomDisplay.getClassNameForHeader(obj);
      emptyHeader = [dimstr,' ',className,' with no employee information'];
      header = sprintf('%s\n',emptyHeader);
      disp(header)
   end
end
```

For example, an empty `EmployeeInfo` object displays like this:

```
Empt = EmployeeInfo.empty(0,5)

Empt =

0x5 EmployeeInfo with no employee information
```

Complete Class Listing

```
classdef EmployeeInfo < handle & matlab.mixin.CustomDisplay
   properties
      Name
      JobTitle
      Department
      Salary
      Password
   end
   methods
      function obj = EmployeeInfo
         obj.Name       = input('Name: ');
         obj.JobTitle   = input('Job Title: ');
         obj.Department = input('Department: ');
         obj.Salary     = input('Salary: ');
         obj.Password   = input('Password: ');
      end
   end
   methods (Access = protected)
      function displayNonScalarObject(objAry)
         dimStr = matlab.mixin.CustomDisplay.convertDimensionsToString(objAry);
         cName = matlab.mixin.CustomDisplay.getClassNameForHeader(objAry);
         headerStr = [dimStr,' ',cName,' members:'];
         header = sprintf('%s\n',headerStr);
         disp(header)
         for ix = 1:length(objAry)
            o = objAry(ix);
            if ~isvalid(o)
               str1 = matlab.mixin.CustomDisplay.getDeletedHandleText;
               str2 = matlab.mixin.CustomDisplay.getClassNameForHeader(o);
               headerInv = [str1,' ',str2];
               tmpStr = [num2str(ix),'. ',headerInv];
               numStr = sprintf('%s\n',tmpStr);
               disp(numStr)
```

```
            else
               numStr = [num2str(ix),'. Employee'];
               disp(numStr)
               propList = struct('Name',o.Name,...
                  'Department',o.Department);
               propgrp = matlab.mixin.util.PropertyGroup(propList);
               matlab.mixin.CustomDisplay.displayPropertyGroups(o,propgrp);
            end
         end
      end

      function displayEmptyObject(obj)
         dimstr = matlab.mixin.CustomDisplay.convertDimensionsToString(obj);
         className = matlab.mixin.CustomDisplay.getClassNameForHeader(obj);
         emptyHeader = [dimstr,' ',className,' with no employee information'];
         header = sprintf('%s\n',emptyHeader);
         disp(header)
      end
   end
end
```

See Also

Related Examples

- “Choose a Technique for Display Customization” on page 18-15

Overloading the disp Function

In this section...

Display Methods

Subclassing `matlab.mixin.CustomDisplay` is the best approach to customizing object display. However, if you do not derive your class from `matlab.mixin.CustomDisplay`, overload the `disp` function to change how MATLAB displays objects of your class.

MATLAB calls the `display` function whenever an object is referred to in a statement that is not terminated by a semicolon. For example, the following statement creates the variable `a`. MATLAB calls `display`, which displays the value of `a` in the command line.

```
a = 5
```

```
a =
     5
```

`display` then calls `disp`.

Overloaded disp

The built-in `display` function prints the name of the variable that is being displayed, if an assignment is made, or otherwise uses `ans` as the variable name. Then `display` calls `disp` to handle the actual display of the values.

If the variable that is being displayed is an object of a class that overloads `disp`, then MATLAB always calls the overloaded method. MATLAB calls `display` with two arguments and passes the variable name as the second argument.

Relationship Between disp and display

MATLAB invokes the built-in `display` function when the following occur:

- MATLAB executes a statement that returns a value and is not terminated with a semicolon.
- There is no left-side variable, then MATLAB prints `ans =` followed by the value.
- Code explicitly invokes the `display` function.

When invoking `display`:

- If the input argument is an existing variable, `display` prints the variable name and equal sign, followed by the value.
- If the input is the result of an expression, display does not print `ans =`.

MATLAB invokes the built-in `disp` function when the following occurs:

- The built-in `display` function calls `disp`.
- Code explicitly invokes `disp`.

For empty built-in types (numeric types, `char`, `struct`, and `cell`) the `display` function displays:

- `[]` — for numeric types
- `"0x0 struct array with no fields."` — for empty `structs`.
- `"0x0 empty cell array"` — for empty cell arrays.
- `"0x0 empty char array"` — for empty `char` arrays
- `"0x0 empty string array"` — for empty `string` arrays

`disp` differs from `display` in these ways:

- `disp` does not print the variable name or `ans`.
- `disp` prints nothing for built-in types (numeric types, `char`, `struct`, and `cell`) when the value is empty.

See Also

Related Examples

- “Custom Display Interface” on page 18-2
- “Overload disp for DocPolynom” on page 19-11

Custom Compact Display Interface

In addition to the custom display options described in “Custom Display Interface” on page 18-2, you can also customize the way MATLAB displays objects in compact display scenarios. Compact display scenarios occur when an object array is inside a container, such as a structure, cell array, or table. You can customize the way object arrays are displayed in a compact display by inheriting from `matlab.mixin.CustomCompactDisplayProvider`.

(Since R022b) The rules implemented by `CustomCompactDisplayProvider` are honored by the Live Editor, the Variables editor in MATLAB Online, and the single-line `'Value'` field in the Workspace panel in MATLAB Online.

Customization Options Available for Compact Display

`CustomCompactDisplayProvider` provides customization options in two main areas of compact displays:

- **Single-line layout** — The display of the object array is confined to a single line. This display layout occurs when the object array is contained within a structure, cell array, or property of a MATLAB object. Override the `CustomCompactDisplayProvider` method `compactRepresentationForSingleLine` to customize this layout.
- **Columnar layout** — The object array is displayed in multiple rows as part of column-oriented or tabular data. This display layout occurs when the object array is contained within a table variable. Override the `CustomCompactDisplayProvider` method `compactRepresentationForColumn` to customize this layout.

Three additional `CustomCompactDisplayProvider` utility methods enable you to control how much of the data is used to construct a display of the object array:

- `partialDataRepresentation` — The compact display of the object array is constructed from only the leading elements of the data set or only the leading and trailing elements, omitting the values in between.
- `fullDataRepresentation` — The compact display of the object array is constructed based on the entire set of data.
- `widthConstrainedDataRepresentation` — The compact display of the object array is constructed from the data based on a width constraint you provide.

Each of these three utility methods also provides an `Annotation` name-value argument, which enables you to include a row vector of strings to serve as descriptors in your compact displays. For example, in the case of a class representing polynomials, adding an annotation can be useful when only the dimensions and class of the data fit in the display:

```
struct with fields:

   prop1: [1x3 sym] (Third-order polynomial)
```

Designing a Class with a Customized Compact Display

When considering how to customize the compact display for a class, first write the class without any customization and test to see if the default display options need to be changed for your application. For example, this class stores a single polynomial as a string and its order as a `double`.

```
classdef Polynomial
    properties
        polynomialString string {mustBeScalarOrEmpty};
        polynomialOrder double {mustBeScalarOrEmpty};
    end
    methods
        function obj = Polynomial(polyString,polyOrder)
            obj.polynomialString = polyString;
            obj.polynomialOrder = polyOrder;
        end
    end
end
```

The full code of this class with customized compact display functionality is available at the end of the example.

Create an instance of `Polynomial`. Display the instance in a cell array to check the single-line layout.

```
poly = "(x^10)/100 + 300*x^9 + 200*x^8 + 4000*x^7 + 600*x^6 + 700*x^5 + 45*x^4 + 90*x^3 + 4*x^2 +
a = Polynomial(poly,10);
{a}
```

```
ans =

  1×1 cell array

    {1×1 Polynomial}
```

The single-line layout does not provide users with much information. Change `Polynomial` to inherit from `matlab.mixin.CustomCompactDisplayProvider` and override the `compactRepresentationForSingleLine` method. Call the utility method `widthConstrainedDataRepresentation` with `AllowTruncatedDisplayForScalar` set to `true`. MATLAB now truncates the polynomial string based on the width available.

```
function displayRep = compactRepresentationForSingleLine(obj,displayConfiguration,width)
   displayRep = widthConstrainedDataRepresentation(obj,displayConfiguration,width,...
      StringArray=obj.polynomialString,AllowTruncatedDisplayForScalar=true);
end
```

Now, the display of the instance in a cell array contains as much of the string as possible.

```
{a}
```

```
ans =

  1×1 cell array

    {[(x^10)/100 + 300*x^9 + 200*x^8 + 4000*x^7 + 600*x^6 + 700*x^5 + 45*x^4 + 90*x^3 + 4*x^2 + 1
```

For a table display, the default shows only the class and dimensions, so override `compactRepresentationForColumn`. The method returns an instance of `matlab.display.DimensionsAndClassNameRepresentation` with an annotation, which in this case identifies the order of the polynomial. To cover the case when a table has a row vector of `Polynomial` instances in a single entry, the method reshapes the `polynomialOrder` property to match the shape of the instance and then finds the maximum order in that row for display.

```
function displayRep = compactRepresentationForColumn(obj,displayConfig,width)
   import matlab.display.DimensionsAndClassNameRepresentation;
```

```
    polOrder = reshape([obj.polynomialOrder],size(obj));
    maxPolOrderPerRow = max(polOrder,[],2);
    annotationStr = "Polynomial of order " + string(maxPolOrderPerRow);
    displayRep = DimensionsAndClassNameRepresentation(obj,...
        displayConfig,Annotation=annotationStr);
end
```

With this code in place, you can create a table of estimation results, for example, that identifies the order of the polynomial. Create a second `Polynomial`, `b`. Then create two tables, one with `a` and `b` in a column, and one with `a` and `b` in the same row.

```
b = Polynomial("x^3 + x^2 + x",3);
Result = [1;2];
Estimate = [a;b];
T = table(Result,Estimate)
```

```
T =

  2×2 table

    Result                    Estimate
    ______    ____________________________________________

      1       1×1 Polynomial  (Polynomial of order 10)
      2       1×1 Polynomial  (Polynomial of order 3)
```

With `a` and `b` in the same row, only the maximum order is displayed.

```
Result = 1;
Estimate = [a b];
T1 = table(Result,Estimate)
```

```
T1 =

  1×2 table

    Result                    Estimate
    ______    ____________________________________________

       1      1×2 Polynomial  (Polynomial of order 10)
```

Complete Polynomial Class

```
classdef Polynomial < matlab.mixin.CustomCompactDisplayProvider
    properties
        polynomialString string {mustBeScalarOrEmpty};
        polynomialOrder double {mustBeScalarOrEmpty};
    end
    methods
        function obj = Polynomial(polyString,polyOrder)
            obj.polynomialString = polyString;
            obj.polynomialOrder = polyOrder;
        end

        function displayRep = compactRepresentationForSingleLine(obj,displayConfiguration,width)
            displayRep = widthConstrainedDataRepresentation(obj,displayConfiguration,width,...
                StringArray=obj.polynomialString,AllowTruncatedDisplayForScalar=true);
        end

        function displayRep = compactRepresentationForColumn(obj,displayConfig,width)
            import matlab.display.DimensionsAndClassNameRepresentation;
            polOrder = reshape([obj.polynomialOrder],size(obj));
            maxPolOrderPerRow = max(polOrder,[],2);
```

```
            annotationStr = "Polynomial of order " + string(maxPolOrderPerRow);
            displayRep = DimensionsAndClassNameRepresentation(obj,...
                displayConfig,Annotation=annotationStr);
        end
    end
end
```

See Also

Related Examples

- `matlab.mixin.CustomCompactDisplayProvider`

Defining Custom Data Types

Representing Polynomials with Classes

In this section...

You can use classes to define new data types. This example implements a class that represents polynomials. The class stores the coefficients of the polynomial terms in a vector and overrides the default MATLAB display to show the polynomials as powers of *x*. Using customized indexing, the class also enables you to evaluate the polynomials at one or more values of *x* using parentheses indexing syntax.

Class Requirements

The design requirements for the `DocPolynom` class are:

- Value class behavior —Behave like MATLAB numeric variables when copied and passed to functions.
- Scalar object behavior — Polynomial objects cannot be concatenated, and polynomial array size must always be (1,1).
- Customized indexing behavior — Evaluate a polynomial using parentheses indexing syntax. `p(x)` evaluates the polynomial represented by object `p` at each value in `x`.
- Specialized display — Use the coefficients stored in the polynomial object to display the polynomial as an algebraic expression.
- Override addition, subtraction, and multiplication — Adding, subtracting, or multiplying polynomial objects returns the result of the corresponding algebraic operation on the two polynomials.
- Double converter — Convert a polynomial object to a `double` array so it can be used with existing MATLAB functions that accept numeric inputs.

DocPolynom Class Members

The class defines the property `coef` for storage of the polynomial coefficients.

`DocPolynom` Class Properties

Name	Class	Default	Description
`coef`	`double`	`[]`	Vector of polynomial coefficients, in order of the highest exponent of x to lowest.

This table summarizes the methods for the `DocPolynom` class.

`DocPolynom` Class Methods

Name	Description
`DocPolynom`	Class constructor
`double`	Converts the `DocPolynom` object to a double. In other words, this method returns the coefficients in a vector of `double` values.
`char`	Creates a formatted display of the `DocPolynom` object as powers of x. This method is used by the `disp` method.
`disp`	Defines how MATLAB displays `DocPolynom` objects on the command line.
`plus`	Adds `DocPolynom` objects.
`minus`	Subtracts `DocPolynom` objects.
`mtimes`	Multiplies `DocPolynom` objects.
`dispPoly`	Evaluates the polynomial for one or more values and returns the results in an organized list instead of a vector of `double` values.
`parenReference`	Enables evaluation of a polynomial using parentheses indexing syntax. `p(x)` evaluates the polynomial object `p` at each value in `x`.

Using the DocPolynom Class

These examples show the basic use of the `DocPolynom` class. Create `DocPolynom` objects to represent $f(x) = x^3 - 2x - 5$ and $f(x) = 2x^4 + 3x^2 + 2x - 7$.

```
p1 = DocPolynom([1 0 -2 -5])
```

```
p1 =
   x^3 - 2*x - 5
```

```
p2 = DocPolynom([2 0 3 2 -7])
```

```
p2 =
   2*x^4 + 3*x^2 + 2*x - 7
```

Find the roots of the polynomial `p1`. Use the `double` method of the object and pass the result to the `roots` function.

```
roots(double(p1))
```

```
ans =

   2.0946 + 0.0000i
  -1.0473 + 1.1359i
  -1.0473 - 1.1359i
```

Add the two polynomials `p1` and `p2`. MATLAB calls the `plus` method defined for the `DocPolynom` class when you add two `DocPolynom` objects.

```
p1 + p2

ans =

2*x^4 + x^3 + 3*x^2 - 12
```

DocPolynom Class Synopsis

Class Code	Description
`classdef DocPolynom < matlab.mixin.Scalar`	Value class that implements a data type for polynomials. The class inherits from `matlab.mixin.Scalar`, which enforces scalar class behavior and also provides functionality to customize parentheses indexing. For more information on the superclass, see `matlab.mixin.Scalar`.

`properties`
`  coef`
`end`
	Vector of polynomial coefficients.

`methods`
`  function obj = DocPolynom(c)`
`  if nargin > 0`
`  if isa(c,'DocPolynom')`
`  obj.coef = c.coef;`
`  else`
`  obj.coef = c(:).';`
`  end`
`  end`
`end`
	Class constructor that creates objects using either:
	• An existing `DocPolynom` object
	• A vector of doubles
	For more information, see “The DocPolynom Constructor” on page 19-9.

`function obj = set.coef(obj,val)`
`  if ~isa(val,'double')`
`  error('Coefficients must be doubles.')`
`  end`
`  ind = find(val(:).'~=0);`
`  if isempty(ind)`
`  obj.coef = val;`
`  else`
`  obj.coef = val(ind(1):end);`
`  end`
`end`
	Set method for `coef` property:
	• Restricts coefficients to type double
	• Removes leading zeros from the coefficient vector
	For more information, see “Remove Leading Zeros” on page 19-9.

`function c = double(obj)`
`  c = obj.coef;`
`end`
	Convert `DocPolynom` object to `double` by returning the coefficients.
	For more information, see “Convert DocPolynom Objects to Other Classes” on page 19-10.

Class Code	Description

`function str = char(obj)`
`  if all(obj.coef == 0)`
`  s = '0';`
`  str = s;`
`  return`
`  else`
`  d = length(obj.coef) - 1;`
`  s = cell(1,d);`
`  ind = 1;`
`  for a = obj.coef`
`  if a ~= 0`
`  if ind ~= 1`
`  if a > 0`
`  s(ind) = {' + '};`
`  ind = ind + 1;`
`  else`
`  s(ind) = {' - '};`
`  a = -a;`
`  ind = ind + 1;`
`  end`
`  end`
`  if a ~= 1 || d == 0`
`  if a == -1`
`  s(ind) = {'-'};`
`  ind = ind + 1;`
`  else`
`  s(ind) = {num2str(a)};`
`  ind = ind + 1;`
`  if d > 0`
`  s(ind) = {'*'};`
`  ind = ind + 1;`
`  end`
`  end`
`  end`
`  if d >= 2`
`  s(ind) = {['x^' int2str(d)]};`
`  ind = ind + 1;`
`  elseif d == 1`
`  s(ind) = {'x'};`
`  ind = ind + 1;`
`  end`
`  end`
`  d = d - 1;`
`  end`
`  end`
`  str = [s{:}];`
`end`
	Convert `DocPolynom` object to `char`. For more information, see “Convert DocPolynom Objects to Other Classes” on page 19-10.

`function disp(obj)`
`  c = char(obj);`
`  if iscell(c)`
`  disp(['  ' c{:}])`
`  else`
`  disp(c)`
`  end`
`end`
	Overload `disp` function. This method displays objects as output of the `char` method.
	
	For more information, see “Overload disp for DocPolynom” on page 19-11.

Class Code	Description

`function dispPoly(obj,x)`
`  p = char(obj);`
`  y = zeros(length(x));`
`  disp(['f(x) = ',p])`
`  for k = 1:length(x)`
`  y(k) = polyval(obj.coef,x(k));`
`  disp(['  f(',num2str(x(k)),') = ',num2str(y(k))])`
`  end`
`end`
	Return evaluated polynomial with formatted output. This method uses `polyval` in a loop to evaluate the polynomial at specified values of the independent variable.
	For more information, see “Display Evaluated Expression” on page 19-11.

`function r = plus(obj1,obj2)`
`  obj1 = DocPolynom(obj1);`
`  obj2 = DocPolynom(obj2);`
`  k = length(obj2.coef) - length(obj1.coef);`
`  zp = zeros(1,k);`
`  zm = zeros(1,-k);`
`  r = DocPolynom([zp,obj1.coef] + [zm,obj2.coef]);`
`end`
 	
`function r = minus(obj1,obj2)`
`  obj1 = DocPolynom(obj1);`
`  obj2 = DocPolynom(obj2);`
`  k = length(obj2.coef) - length(obj1.coef);`
`  zp = zeros(1,k);`
`  zm = zeros(1,-k);`
`  r = DocPolynom([zp,obj1.coef] - [zm,obj2.coef]);`
`end`
 	
`function r = mtimes(obj1,obj2)`
`  obj1 = DocPolynom(obj1);`
`  obj2 = DocPolynom(obj2);`
`  r = DocPolynom(conv(obj1.coef,obj2.coef));`
`end`
`end`
	Define three arithmetic operators:
	• Polynomial addition
	• Polynomial subtraction
	• Polynomial multiplication
	For information about this code, see “Define Arithmetic Operators” on page 19-12.
	For general information about defining operators, see “Operator Overloading” on page 17-19.

`methods (Access = protected)`
`  function f = parenReference(obj,indexOp)`
`  n = cell2mat(indexOp(1).Indices);`
`  if numel(indexOp) == 1`
`  f = polyval(obj.coef,n);`
`  else`
`  f = polyval(obj.coef,n).(indexOp(2:end));`
`  end`
`  end`
`end`
`end`
	Customize parentheses reference for `DocPolynom` objects. This implementation of `parenReference` enables users to evaluate a polynomial using parentheses indexing syntax. `p(x)` evaluates the polynomial represented by object `p` for each value in `x`.
	For more information, see “Redefine Parentheses Indexing” on page 19-13.

Expand for Class Code

```
classdef DocPolynom < matlab.mixin.Scalar

   properties
      coef
   end
```

```
methods
   function obj = DocPolynom(c)
      if nargin > 0
         if isa(c,'DocPolynom')
            obj.coef = c.coef;
         else
            obj.coef = c(:).';
         end
      end
   end

   function obj = set.coef(obj,val)
      if ~isa(val,'double')
         error('Coefficients must be doubles.')
      end
      ind = find(val(:).'~=0);
      if isempty(ind)
         obj.coef = val;
      else
         obj.coef = val(ind(1):end);
      end
   end

   function c = double(obj)
      c = obj.coef;
   end

   function str = char(obj)
      if all(obj.coef == 0)
         s = '0';
         str = s;
         return
      else
         d = length(obj.coef) - 1;
         s = cell(1,d);
         ind = 1;
         for a = obj.coef
            if a ~= 0
               if ind ~= 1
                  if a > 0
                     s(ind) = {' + '};
                     ind = ind + 1;
                  else
                     s(ind) = {' - '};
                     a = -a;
                     ind = ind + 1;
                  end
               end
               if a ~= 1 || d == 0
                  if a == -1
                     s(ind) = {'-'};
                     ind = ind + 1;
                  else
                     s(ind) = {num2str(a)};
                     ind = ind + 1;
                     if d > 0
                        s(ind) = {'*'};
```

```
                    ind = ind + 1;
                end
            end
        end
        if d >= 2
            s(ind) = {['x^' int2str(d)]};
            ind = ind + 1;
        elseif d == 1
            s(ind) = {'x'};
            ind = ind + 1;
        end
    end
    d = d - 1;
  end
end
str = [s{:}];
end

function disp(obj)
c = char(obj);
if iscell(c)
   disp(['     ' c{:}])
else
   disp(c)
end
end

function dispPoly(obj,x)
p = char(obj);
y = zeros(length(x));
disp(['f(x) = ',p])
for k = 1:length(x)
   y(k) = polyval(obj.coef,x(k));
   disp(['   f(',num2str(x(k)),') = ',num2str(y(k))])
end
end

function r = plus(obj1,obj2)
obj1 = DocPolynom(obj1);
obj2 = DocPolynom(obj2);
k = length(obj2.coef) - length(obj1.coef);
zp = zeros(1,k);
zm = zeros(1,-k);
r = DocPolynom([zp,obj1.coef] + [zm,obj2.coef]);
end

function r = minus(obj1,obj2)
obj1 = DocPolynom(obj1);
obj2 = DocPolynom(obj2);
k = length(obj2.coef) - length(obj1.coef);
zp = zeros(1,k);
zm = zeros(1,-k);
r = DocPolynom([zp,obj1.coef] - [zm,obj2.coef]);
end

function r = mtimes(obj1,obj2)
obj1 = DocPolynom(obj1);
obj2 = DocPolynom(obj2);
```

```
            r = DocPolynom(conv(obj1.coef,obj2.coef));
         end
      end

      methods (Access = protected)
         function f = parenReference(obj,indexOp)
            n = cell2mat(indexOp(1).Indices);
            if numel(indexOp) == 1
               f = polyval(obj.coef,n);
            else
               f = polyval(obj.coef,n).(indexOp(2:end));
            end
          end
      end
   end
```

The DocPolynom Constructor

This is the `DocPolynom` class constructor:

```
function obj = DocPolynom(c)
   if nargin > 0
      if isa(c,'DocPolynom')
         obj.coef = c.coef;
       else
          obj.coef = c(:).';
       end
    end
end
```

Constructor Calling Syntax

The `DocPolynom` constructor can accept two different input arguments:

- An existing `DocPolynom` object — Calling the constructor with an existing `DocPolynom` object as an input argument returns a new `DocPolynom` object with the same coefficients as the input argument. The `isa` function checks for this input.
- Coefficient vector — When the input argument is not a `DocPolynom` object, the constructor attempts to reshape the values into a row vector and assign them to the `coef` property.

  The `coef` property set method restricts property values to doubles. See “Remove Leading Zeros” on page 19-9 for a description of the property set method.

This example uses a vector as the input argument to the `DocPolynom` constructor:

```
p = DocPolynom([1 0 -2 -5])
p =
   x^3 - 2*x -5
```

This statement creates an instance of the `DocPolynom` class with the specified coefficients. The display of the object shows the equivalent polynomial using MATLAB language syntax. The `DocPolynom` class implements this display using the `disp` and `char` class methods.

Remove Leading Zeros

The `DocPolynom` class represents polynomials as row vectors containing coefficients ordered by descending powers. Zeros in the coefficient vector represent terms that are not in the polynomial.

Leading zeros, therefore, can be ignored when forming the polynomial. In fact, some `DocPolynom` class methods use the length of the coefficient vector to determine the degree of the polynomials, so removing leading zeros from the coefficient vector ensures that the vector length represents the correct polynomial degree.

The `DocPolynom` class stores the coefficient vector in a property that uses a set method to remove leading zeros from the specified coefficients before setting the property value.

```
function obj = set.coef(obj,val)
   if ~isa(val,'double')
     error('Coefficients must be doubles.')
   end
   ind = find(val(:).'~=0);
   if isempty(ind)
     obj.coef = val;
   else
      obj.coef = val(ind(1):end);
   end
end
```

Convert DocPolynom Objects to Other Classes

The `DocPolynom` class defines two methods to convert `DocPolynom` objects to other classes:

- `double` — Converts to the double numeric type so functions can perform mathematical operations on the coefficients.
- `char` — Converts to characters used to format output for display in the Command Window.

The Double Converter

The `double` converter method for the `DocPolynom` class returns the coefficient vector:

```
function c = double(obj)
    c = obj.coef;
end
```

For the `DocPolynom` object `p`, `double` returns a vector of class `double`.

```
p = DocPolynom([1 0 -2 -5]);
c = double(p)

c =

     1     0    -2    -5
```

The Character Converter

The `char` method returns a `char` vector that represents the polynomial displayed as powers of `x`. The `char` vector returned is a syntactically correct MATLAB expression.

The `char` method uses a cell array to collect the `char` vector components that make up the displayed polynomial. The `disp` method uses the `char` method to format the `DocPolynom` object for display. Users of `DocPolynom` objects are not likely to call the `char` or `disp` methods directly, but these methods enable the `DocPolynom` class to behave like other data classes in MATLAB.

Overload disp for DocPolynom

To provide a more useful display of `DocPolynom` objects, this class overloads `disp` in the class definition. This `disp` method relies on the `char` method to produce a text representation of the polynomial, which it then displays.

The `char` method returns a cell array or the character `'0'` if the coefficients are all zero.

```
function disp(obj)
   c = char(obj);
   if iscell(c)
      disp(['     ' c{:}])
   else
      disp(c)
   end
end
```

When MATLAB Calls the disp Method

This statement creates a `DocPolynom` object. Because the statement is not terminated with a semicolon, the resulting output is displayed on the command line using the overloaded `disp` method.

```
p = DocPolynom([1 0 -2 -5])
```

```
p =
    x^3 - 2*x - 5
```

Display Evaluated Expression

The `dispPoly` method evaluates the polynomial for one or more values of `x`. The method loops through the input values of `x` and uses the `polyval` function with the `coef` property to evaluate the polynomial.

```
function dispPoly(obj,x)
   p = char(obj);
   y = zeros(length(x));
   disp(['f(x) = ',p,])
   for k = 1:length(x)
      y(k) = polyval(obj.coef,x(k));
      disp(['   f(',num2str(x(k)),') = ',num2str(y(k))])
    end
end
```

Create a `DocPolynom` object `p`:

```
p = DocPolynom([1 0 -2 -5])
```

```
p =
   x^3 - 2*x - 5
```

Evaluate the polynomial at three values of `x`, `[3 5 9]`. Instead of returning a vector of values, the method uses function notation to present the results in an organized list.

```
dispPoly(p,[3 5 9])
```

```
f(x) = x^3 - 2*x - 5
   f(3) = 16
```

```
f(5) = 110
f(9) = 706
```

Define Arithmetic Operators

The `DocPolynom` class implements methods for three arithmetic operations.

Method and Syntax	Operation
`plus(a,b)`	Addition
`minus(a,b)`	Subtraction
`mtimes(a,b)`	Matrix multiplication

The overloaded `plus`, `minus`, and `mtimes` methods accept argument pairs that include at least one `DocPolynom` object.

Define the + Operator

If either `p` or `q` is a `DocPolynom` object, this expression generates a call to the `plus` method overload defined by `DocPolynom` unless the other object is of higher precedence.

```
p + q
```

This method overloads the `plus` (`+`) operator for the `DocPolynom` class.

```
function r = plus(obj1,obj2)
   obj1 = DocPolynom(obj1);
   obj2 = DocPolynom(obj2);
   k = length(obj2.coef) - length(obj1.coef);
   zp = zeros(1,k);
   zm = zeros(1,-k);
   r = DocPolynom([zp,obj1.coef] + [zm,obj2.coef]);
end
```

The `plus` method performs these actions:

- Ensure that both input arguments are `DocPolynom` objects so that expressions that involve a `DocPolynom` and a `double` work correctly.
- Access the two coefficient vectors and, if necessary, pad one of them with zeros to make both the same length. The actual addition is simply the vector sum of the two coefficient vectors.
- Call the `DocPolynom` constructor to create a properly typed object that is the result of adding the polynomials.

Define the - Operator

The `minus` operator (`-`) uses the same approach as the `plus` (`+`) operator. The `minus` method computes `p - q`. The dominant argument must be a `DocPolynom` object.

```
function r = minus(obj1,obj2)
   obj1 = DocPolynom(obj1);
   obj2 = DocPolynom(obj2);
   k = length(obj2.coef) - length(obj1.coef);
   zp = zeros(1,k);
   zm = zeros(1,-k);
   r = DocPolynom([zp,obj1.coef] - [zm,obj2.coef]);
end
```

Define the * Operator

The `mtimes` method computes the product `p*q`. The `mtimes` method implements *matrix* multiplication because the multiplication of two polynomials is the convolution (`conv`) of their coefficient vectors:

```
methods
   function r = mtimes(obj1,obj2)
      obj1 = DocPolynom(obj1);
      obj2 = DocPolynom(obj2);
      r = DocPolynom(conv(obj1.coef,obj2.coef));
   end
end
```

Using the Arithmetic Operators

Create a `DocPolynom` object.

```
p = DocPolynom([1 0 -2 -5]);
```

These two arithmetic operations call the `DocPolynom` `plus` and `mtimes` methods.

```
q = p + 1
r = p*q
```

```
q =
    x^3 - 2*x - 4

r =
    x^6 - 4*x^4 - 9*x^3 + 4*x^2 + 18*x + 20
```

Redefine Parentheses Indexing

The `DocPolynom` class inherits from `matlab.mixin.Scalar`, which in turn inherits from the modular indexing class `matlab.mixin.indexing.RedefinesParen`. Overloading the `parenReference` method of `RedefinesParen` enables users to evaluate a polynomial represented by a `DocPolynom` object using parentheses indexing syntax.

For example, create a `DocPolynom` object `p`.

```
p = DocPolynom([1 0 -2 -5])
```

```
p =
    x^3 - 2*x - 5
```

The overloaded `parenReference` method evaluates the value of the polynomial at `x = 3` and at `x = 4` using this command.

```
p([3 4])
```

```
ans =
    16   51
```

Modular Indexing Implementation Details

The `parenReference` method handles expressions of the form `p(x)`, where `p` is a `DocPolynom` object and `x` contains numeric inputs. Instead of a traditional MATLAB indexing operation, however,

`parenReference` uses `polyval` to evaluate the polynomial using the coefficients stored in the `coef` property.

```
methods (Access = protected)
    function f = parenReference(obj,indexOp)
       n = cell2mat(indexOp(1).Indices);
       if numel(indexOp) == 1
          f = polyval(obj.coef,n);
       else
          f = polyval(obj.coef,n).(indexOp(2:end));
       end
    end
end
```

The method performs these steps:

1. Extract the indexing values from `indexOp`, which is an instance of the `matlab.indexing.IndexingOperation` class. The `indexOp` object stores them as a cell array, and the method converts them to a numeric array and stores them in `n`.
2. Calculate the number of indexing operations in the expression.
3. Evaluate the polynomial at the values in `n`. The intended use of this syntax includes one parentheses indexing operation.
4. If the method finds more than one indexing operation, it uses `polyval` to evaluate the polynomial and then forwards the rest of the indexing operations to MATLAB using the forwarding syntax, `.(indexOp(2:end))`. The class does not support any additional customized indexing operations, so MATLAB returns an error.

   For example, attempting to evaluate a polynomial object while also using dot indexing to try to access its `coef` property at the same time errors.

   ```
   p(5).coef(1)
   ```

   ```
   Dot indexing is not supported for variables of this type.
   ```

   For more information on forwarding operations with modular indexing, see “Forward Indexing Operations” on page 17-30.

Using the modular indexing class in this way means that only parentheses reference operations are customized. Dot access to properties and methods are unaffected and are handled by MATLAB as expected.

Note The `matlab.mixin.Scalar` and `matlab.mixin.indexing.RedefinesParen` functionality was introduced in R2021b.

20

Designing Related Classes

A Class Hierarchy for Heterogeneous Arrays

In this section...

Interfaces Based on Heterogeneous Arrays

A heterogeneous class hierarchy lets you create arrays containing objects of different classes that are related though inheritance. You can define class methods that operate on these heterogeneous arrays as a whole.

A class design based on heterogeneous arrays provides a more convenient interface than, for example, extracting elements from a cell array and operating on these elements individually. For more information on the design of class hierarchies that support heterogeneous arrays, see “Designing Heterogeneous Class Hierarchies” on page 10-22.

All heterogeneous hierarchies derive from `matlab.mixin.Heterogeneous`.

Define Heterogeneous Hierarchy

Note This example does not use valid terminology or techniques for managing financial assets. The purpose of this example is only to illustrate techniques for defining heterogeneous class hierarchies.

This example implements a system of classes to represent financial assets, such as stocks, bonds, and cash. Classes to represent categories of assets have certain common requirements. Each instance has one of the following:

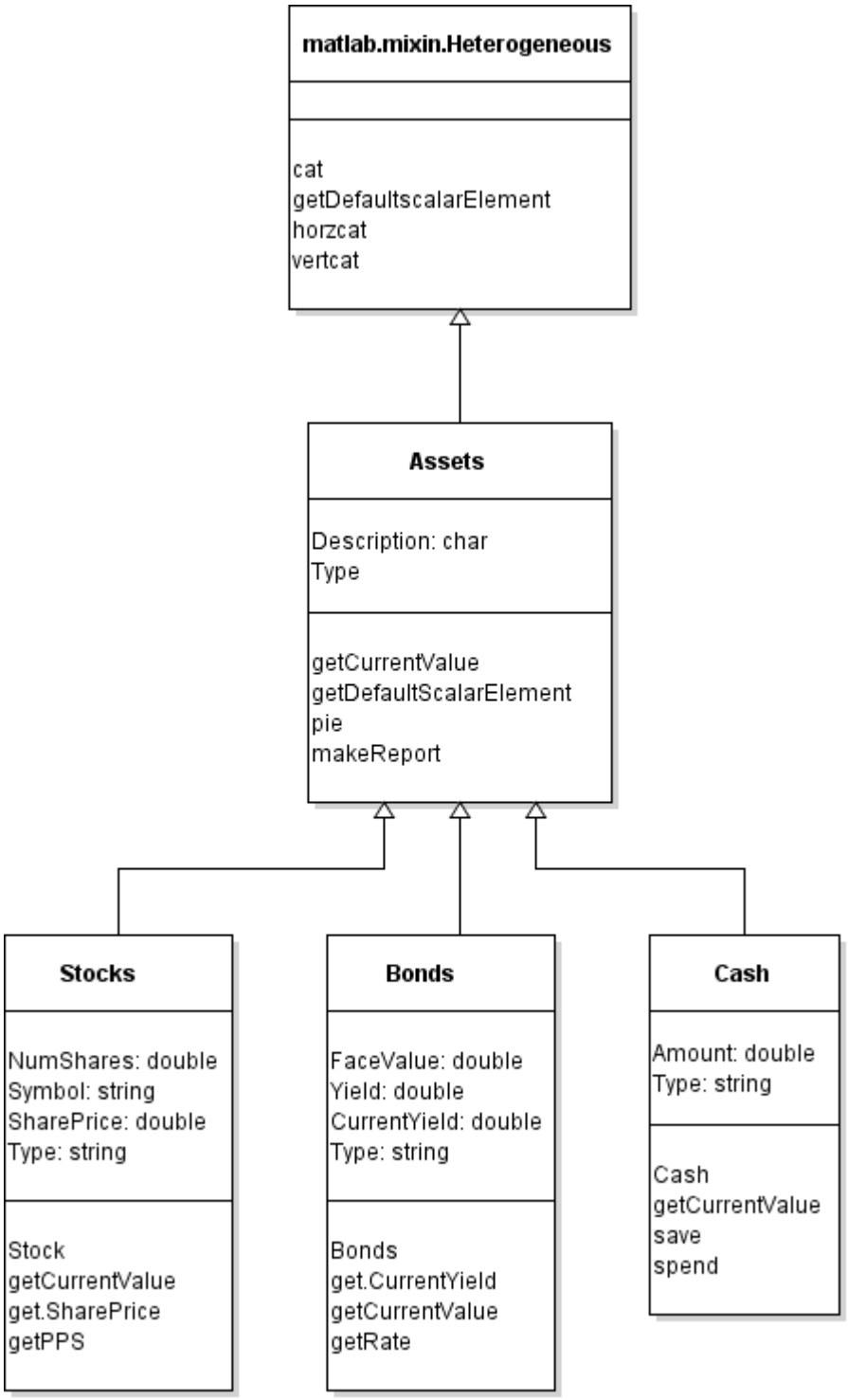
- Textual description
- Type (stock, bond, or cash)
- Means to determine the current value of the asset

Heterogeneous arrays of these objects need methods that can operate on the whole array. These operations include:

- Creating a table of information about all assets contained in the array
- Graphing the relative contribution of each asset type contained in the array

These requirements are factored into the class that is the root of the hierarchy. The root class derives from `matlab.mixin.Heterogeneous`. In the following diagram, the `Assets` class is the root of the

hierarchy. The `Stocks`, `Bonds`, and `Cash` classes provide the specialization required for each type of asset.



Assets Class

The `Assets` class:

- Derives directly from `matlab.mixin.Heterogeneous`
- Is the root of the heterogeneous hierarchy
- Is abstract
- Is the class of heterogeneous arrays composed of any mixture of `Stock`, `Bond`, and `Cash` objects

Properties

The `Assets` class defines two properties:

- `Description` — A general description of the individual asset constrained to be of class `char`.
- `Type` — The type of asset defined as an abstract property that each subclass implements.

Methods

The `Assets` class defines these methods:

- `pie` — A sealed method that creates a pie chart showing the relative mix of asset types.
- `makeReport` — A sealed method that creates a report listing the assets.
- `getCurrentValue` — An abstract method that each concrete subclass must implement to return the current value of the asset.
- `getDefaultScalarElement` — `matlab.mixin.Heterogeneous` class method overridden in the `Assets` class to specify a default object. The `Assets` class is abstract so it cannot be used as the default object. For more information, see “Default Object” on page 20-9.

Methods in Heterogeneous Hierarchies

Methods defined by the `Assets` class are either:

- Concrete methods (fully implemented) that subclasses do not override
- Abstract methods (signatures only) that subclasses implement

Concrete methods defined by superclasses in a heterogeneous hierarchy must specify the `Sealed` attribute. Sealing these methods prevents subclasses from overriding methods implemented by the superclass. When calling methods on a heterogeneous array, MATLAB calls the methods defined by the class of the array (`Assets` in this example).

The `pie` and `makeReport` methods are examples of sealed methods that operate on heterogeneous arrays composed of `Stock`, `Bond`, and `Cash` objects.

Abstract methods defined by the superclasses in a heterogeneous hierarchy must specify the `Abstract` attribute. Defining an abstract method in a superclass ensures that concrete subclasses have an implementation for that exact method name. Use these methods element-wise so that each object calls its own method.

The `getCurrentValue` method is an example of an abstract method that is implemented by each subclass to get the current value of each asset.

Each type of subclass object calculates its current value in a different way. If you add another category of asset by adding another subclass to the hierarchy, this class must implement its own

version of a `getCurrentValue` method. Because all subclasses implement a `getCurrentValue` method, the `pie` and `makeReport` methods work with newly added subclasses.

For more information on the `Sealed` and `Abstract` method attributes, see “Method Attributes” on page 9-4.

Assets Class Code

The `Assets` class and other classes in the hierarchy are contained in a package called `financial`.

```
classdef  Assets < matlab.mixin.Heterogeneous
   % file: +financial/@Assets/Assets.m
   properties
      Description char = 'Assets'
   end
   properties (Abstract, SetAccess = private)
      Type
   end
   methods (Abstract)
      % Not implemented by Assets class
      value = getCurrentValue(obj)
   end
   methods (Static, Sealed, Access = protected)
      function defaultObject = getDefaultScalarElement
         defaultObject = financial.DefaultAsset;
      end
   end
   methods (Sealed)
      % Implemented in separate files
      % +financial/@Assets/pie.m
      % +financial/@Assets/makeReport.m
      pie(assetArray)
      makeReport(assetArray)
   end
end
```

For code listings for `pie` and `makeReport`, see “Operating on an Assets Array” on page 20-11.

Stocks Class

The `Stocks` class represents a specific type of financial asset. It is a concrete class that implements the abstract members defined by the `Assets` class, and defines class properties and methods specific to this type of asset.

Properties

The `Stocks` class defines these properties:

- `NumShares` — The number of shares held for this asset.
- `Symbol` — The ticker symbol corresponding to this stock.
- `Type` — `Stocks` class implementation of the abstract property defined by the `Assets` class. This concrete property must use the same attributes as the abstract version (that is, `SetAccess private`).
- `SharePrice` — Dependent property for the price per share. The `get.SharePrice` method obtains the current share price from web services when this property is queried.

Methods

The Stocks class defines these methods:

- `Stocks` — The constructor assigns property values and supports a default constructor called with no input arguments.
- `getCurrentValue` — This method is the `Stocks` class implementation of the abstract method defined by the `Assets` class. It returns the current value of this asset.
- `get.SharePrice` — The property get method for the dependent `SharePrice` property returns the current share price of this stock. For information on how to access web services from MATLAB, see the `webread` function.

Stocks Class Code

```
classdef Stocks < financial.Assets
   properties
      NumShares double = 0
      Symbol string
   end
   properties (SetAccess = private)
      Type = "Stocks"
   end
   properties (Dependent)
      SharePrice double
   end
   methods
      function sk = Stocks(description,numshares,symbol)
         if nargin == 0
            description = '';
            numshares = 0;
            symbol = '';
         end
         sk.Description = description;
         sk.NumShares = numshares;
         sk.Symbol = symbol;
      end
      function value = getCurrentValue(sk)
         value = sk.NumShares*sk.SharePrice;
      end
      function pps = get.SharePrice(sk)
         % Implement web access to obtain
         % Current price per share
         % Returning dummy value
         pps = 1;
      end
   end
end
```

Bonds Class

The `Bonds` class represents a specific type of financial asset. It is a concrete class that implements the abstract members defined by the `Assets` class and defines class properties and methods specific to this type of asset.

Properties

The `Bonds` class defines these properties:

- `FaceValue` — Face value of the bond.
- `Yield` — Annual interest rate of the bond.
- `Type` — `Bonds` class implementation of the abstract property defined by the `Assets` class. This concrete property must use the same attributes as the abstract version (that is, `SetAccess private`).
- `CurrentYield` — Dependent property for the current yield, The `get.CurrentYield` property get method obtains the value from web services.

Methods

The `Bonds` class defines these methods:

- `Bonds` — The constructor assigns property values and supports a default constructor called with no input arguments.
- `getCurrentVlaue` — This method is the `Bonds` class implementation of the abstract method defined by the `Assets` class. It returns the current value of this asset.
- `get.CurrentYield` — The property get method for the dependent `CurrentYield` property returns the current yield on this bond. For information on how to access web serviced from MATLAB, see the `webread` function.

Bonds Class Code

```
classdef Bonds < financial.Assets
   properties
      FaceValue double = 0
      Yield double = 0
   end
   properties (SetAccess = private)
      Type = "Bonds"
   end
   properties (Dependent)
      CurrentYield double
   end
   methods
      function b = Bonds(description,facevalue,yield)
         if nargin == 0
            description = '';
            facevalue = 0;
            yield = 0;
         end
         b.Description = description;
         b.FaceValue = facevalue;
         b.Yield = yield;
      end
      function mv = getCurrentValue(b)
         y = b.Yield;
         cy = b.CurrentYield;
         if cy <= 0 || y <= 0
            mv = b.FaceValue;
         else
            mv = b.FaceValue*y/cy;
```

```
            end
        end
        function r = get.CurrentYield(b)
            % Implement web access to obtain
            % Current yield for this bond
            % Returning dummy value
            r = 0.24;
        end
    end
end
```

Cash Class

The `Cash` class represents a specific type of financial asset. It is a concrete class that implements the abstract members defined by the `Assets` class and defines class properties and methods specific to this type of asset.

Properties

The `Cash` class defines these properties:

- `Amount` — The amount of cash held in this asset.
- `Type` — `Cash` class implementation of the abstract property defined by the `Assets` class. This concrete property must use the same attributes as the abstract version (that is, `SetAccess private`).

Methods

The `Cash` class defines these methods:

- `Cash` — The constructor assigns property values and supports a default constructor called with no input arguments.
- `getCurrentValue` — This method is the `Cash` class implementation of the abstract method defined by the `Assets` class. It returns the current value of this asset.
- `save` — This method adds the specified amount of cash to the existing amount and returns a new `Cash` object with the current amount.
- `spend` — This method deducts the specified amount from the current amount and returns a new `Cash` object with the current amount.

Cash Class Code

```
classdef Cash < financial.Assets
    properties
        Amount double = 0
    end
    properties (SetAccess = private)
        Type = "Cash"
    end
    methods
        function c = Cash(description,amount)
            if nargin == 0
                description = '';
                amount = 0;
            end
```

```
            c.Description = description;
            c.Amount = amount;
        end
        function value = getCurrentValue(c)
            value = c.Amount;
        end
        function c = save(c,amount)
            newValue = c.Amount + amount;
            c.Amount = newValue;
        end
        function c = spend(c,amount)
            newValue = c.Amount - amount;
            if newValue < 0
                c.Amount = 0;
                disp('Your balance is $0.00')
            else
                c.Amount = newValue;
            end
        end
    end
end
```

Default Object

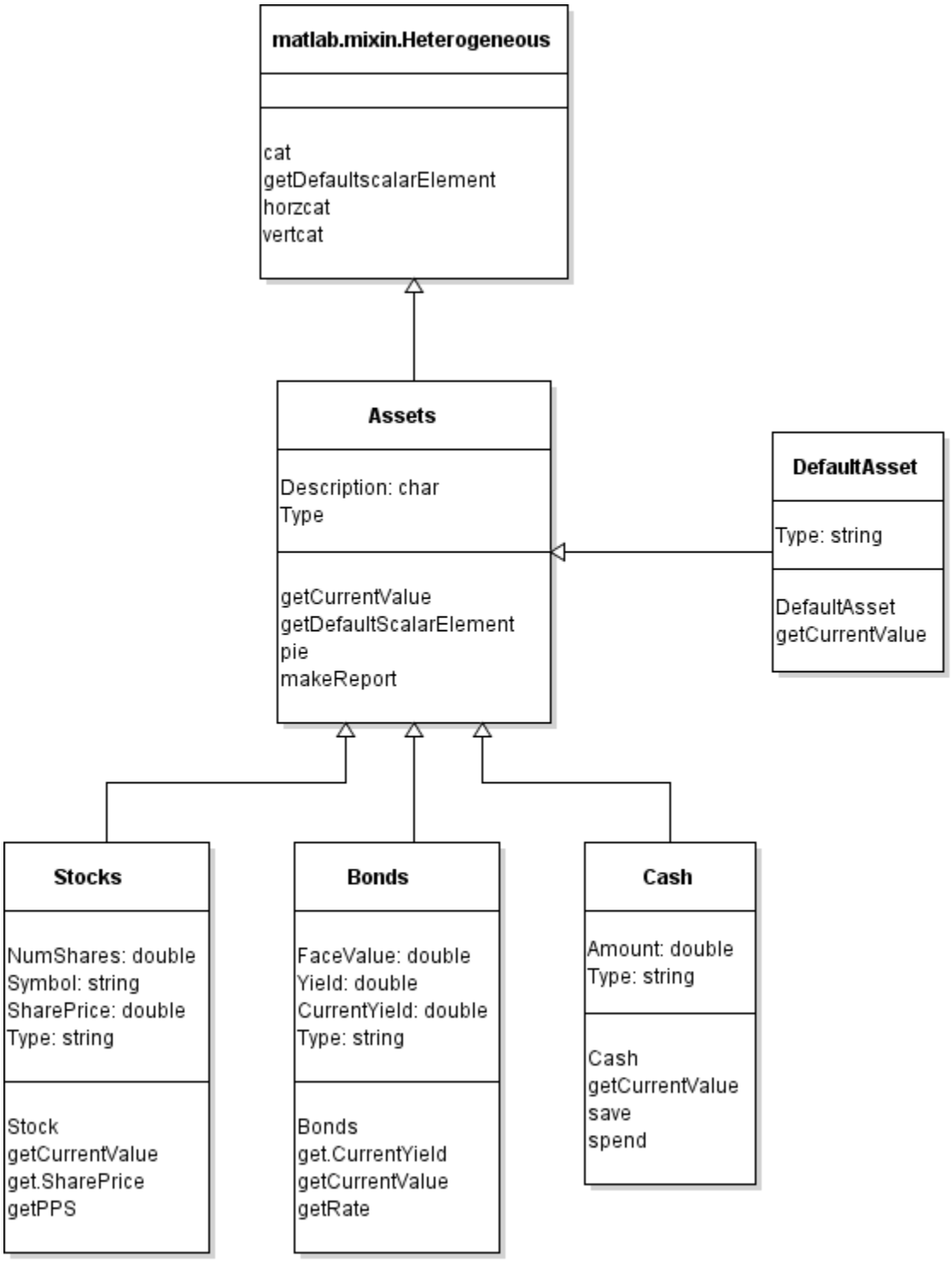
The design of this class hierarchy uses an abstract root class (`Assets`). Therefore, the `Assets` class must specify a concrete class to use as a default object by overriding `getDefaultScalarElement`. In this case, options include:

- Use one of the existing concrete classes for the default object.
- Define a concrete class in the hierarchy to use for the default object.

This implementation adds the `DefaultAsset` class to the hierarchy as a subclass of the `Assets` class. MATLAB creates objects of this class when:

- Creating arrays using indexed assignment with gaps in index numbers
- Loading heterogeneous arrays from MAT-files when MATLAB cannot find the class of an array element.

This diagram shows the addition of the `DefaultAsset` class:



DefaultAsset Class Code

```
classdef DefaultAsset < financial.Assets
   % file: +financial/@DefaultAsset/DefaultAsset.m
   properties (SetAccess = private)
      Type = "DefaultAsset"
```

```
    end
    methods
        function obj = DefaultAsset
            obj.Description = 'Place holder';
        end
        function value = getCurrentValue(~)
            value = 0;
        end
    end
end
```

Operating on an Assets Array

The `Assets` class defines these methods to operate on heterogeneous arrays of asset objects:

- `pie` — Creates a pie chart showing the mix of asset types in the array.
- `makeReport` — Uses the MATLAB `table` object to display a table of asset information.

To operate on a heterogeneous array, a method must be defined for the class of the heterogeneous array and must be sealed. In this case, the class of heterogeneous arrays is always the `Assets` class. MATLAB does not use the class of the individual elements of the heterogeneous array when dispatching to methods.

makeReport Method Code

The `Assets` class `makeReport` method builds a table using the common properties and `getCurrentValue` method for each object in the array.

```
function makeReport(obj)
    numMembers = length(obj);
    descs = cell(1,numMembers);
    types(numMembers) = "";
    values(numMembers) = 0;
    for k = 1:numMembers
        descs{k} = obj(k).Description;
        types(k) = obj(k).Type;
        values(k) = obj(k).getCurrentValue;
    end
    t = table;
    t.Description = descs';
    t.Type = types';
    t.Value = values';
    disp(t)
end
```

The `Assets` class `pie` method calls the `getCurrentValue` method element-wise on objects in the array to obtain the data for the pie chart.

pie Method Code

```
function pie(assetArray)
    stockAmt = 0; bondAmt = 0; cashAmt = 0;
    for k=1:length(assetArray)
        if isa(assetArray(k),'financial.Stocks')
            stockAmt = stockAmt + assetArray(k).getCurrentValue;
        elseif isa(assetArray(k),'financial.Bonds')
```

```
            bondAmt = bondAmt + assetArray(k).getCurrentValue;
         elseif isa(assetArray(k),'financial.Cash')
            cashAmt = cashAmt + assetArray(k).getCurrentValue;
         end
      end
      k = 1;
      if stockAmt ~= 0
         label(k) = {'Stocks'};
         pieVector(k) = stockAmt;
         k = k +1;
      end
      if bondAmt ~= 0
         label(k) = {'Bonds'};
         pieVector(k) = bondAmt;
         k = k +1;
      end
      if cashAmt ~= 0
         label(k) = {'Cash'};
         pieVector(k) = cashAmt;
      end
      pie(pieVector,label)
      tv = stockAmt + bondAmt + cashAmt;
      stg = {['Total Value of Assets: $',num2str(tv,'%0.2f')]};
      title(stg,'FontSize',10)
   end
```

Folders and Files

`Asset` class:

```
+financial/@Assets/Assets.m
+financial/@Assets/makeReport.m
+financial/@Assets/pie.m
```

`Stocks` class:

```
+financial/@Stocks/Stocks.m
```

`Bonds` class:

```
+financial/@Bonds/Bonds.m
```

`Cash` class:

```
+financial/@Cash/Cash.m
```

`DefaultAsset` class:

```
+financial/@DefaultAsset/DefaultAsset.m
```

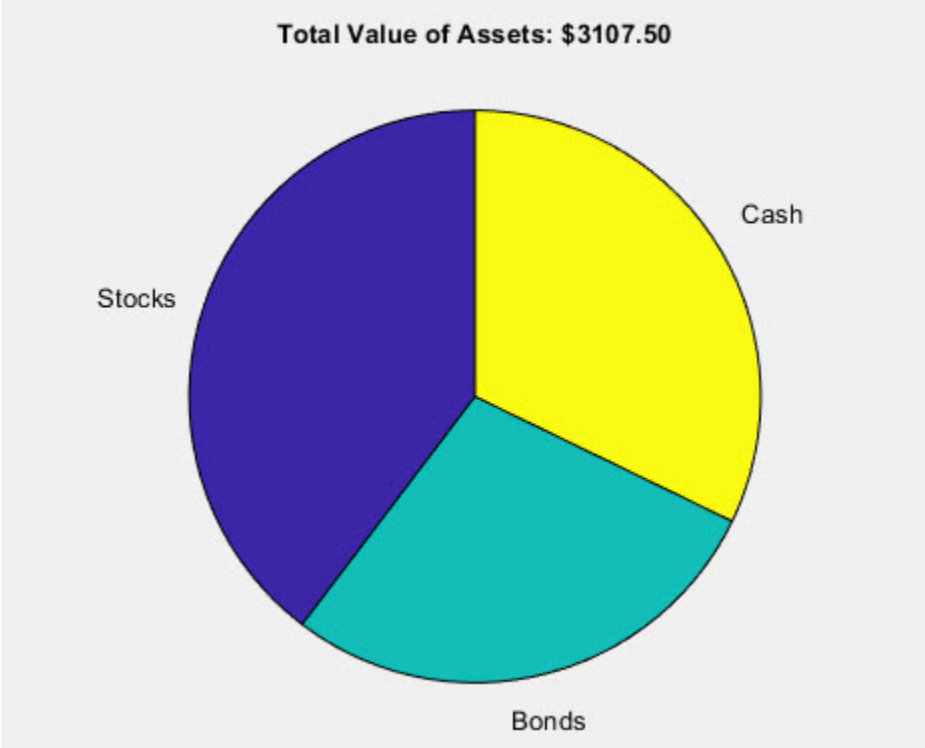
Create an Assets Array

These statements create a heterogeneous array by concatenating the `Stocks`, `Bonds`, and `Cash` objects. Calling the `makeReport` and `pie` methods creates the output shown.

```
s = financial.Stocks('Acme Motor Company',100,string('A'));
b = financial.Bonds('3 Month T',700,0.3);
c(1) = financial.Cash('Bank Account',500);
c(2) = financial.Cash('Gold',500);
```

```
assetArray  = [s,b,c];
makeReport(assetArray)
pie(assetArray)
```

```
        Description           Type      Value
    ______________________    _______    _____

    {'Acme Motor Company'}    "Stock"    1232.5
    {'3 Month T'          }    "Bonds"       875
    {'Bank Account'       }    "Cash"        500
    {'Gold'               }    "Cash"        500
```



See Also

Related Examples

- “Designing Heterogeneous Class Hierarchies” on page 10-22
- “Validate Property Values” on page 8-18
- “Get and Set Methods for Dependent Properties” on page 8-42