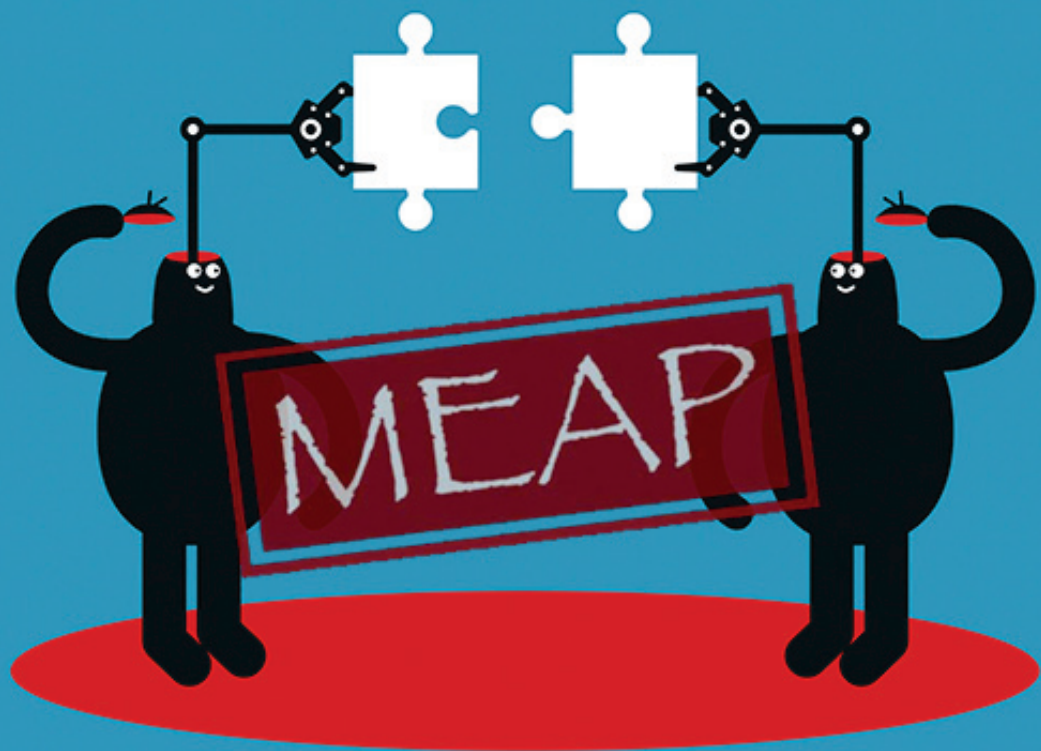


GET PROGRAMMING WITH JAVA



Peggy Fisher

 MANNING



MEAP Edition
Manning Early Access Program
Get Programming with Java
Version 4

Copyright 2019 Manning Publications

For more information on this and other Manning titles go to
manning.com

welcome

First, I want to say thank you for choosing my book, *Get Programming with Java*!

This version of a Java book is designed to increase your knowledge of programming with objects. Since Java is considered an object-oriented programming language, it makes sense that I use that as my tool of choice. It is important to understand that some prior programming experience is essential to get the most out of this book. It is written with the premise that you, the reader, have already learned about basic programming constructs, such as variables, data types, loops and methods. This knowledge can be from programming with JavaScript, Python, or a similar programming language.

The Java programming language is still considered one of the most popular object-oriented programming languages available. It was originally developed under Sun Microsystems, but more recently it was taken over by Oracle.

For this book, you will probably notice right away that I am also a teacher. I approached this book with the same thought and diligence that I use when preparing to teach Java to students. Teaching is my lifelong passion, and I think it comes through in this book. I'm telling you this, so you know this is not strictly a technical book about all things Java, it is a book that teaches you the important concepts in Java starting with an understanding of object representation through advance topics such as inheritance, file processing, collections, and a discussion on design patterns in Java.

After completing this book, you can add Java to the list of tools in your toolbox. In today's job landscape, this is a great addition to any resume. Be sure to update your LinkedIn profile with this skill!

I strongly encourage you to post any questions or comments you have about the content in the [book's forum](#). This feedback is appreciated so that I can make improvements and increase your understanding of the material.

Sincerely,
—Peggy Fisher

brief contents

UNIT 0: GETTING STARTED WITH JAVA

Lesson 1 Get Started

UNIT 1 CLASSES AND OBJECTS

Lesson 2 Creating classes

Lesson 3 Visibility modifiers

Lesson 4 Adding methods

Lesson 5 Adding loops

Lesson 6 Arrays and ArrayLists

Lesson 7 Capstone 1 (Payroll)

UNIT 2: APPLICATION PROGRAMMING INTERFACE (API)

Lesson 8 Standard Java API

Lesson 9 String and StringBuilder Classes

Lesson 10 Static methods and variables

Lesson 11 Using interfaces

Lesson 12 Capstone 2

UNIT 3: PROGRAMMING WITH OBJECTS

Lesson 13 Overloading Methods

Lesson 14 Overriding Methods

Lesson 15 Polymorphism Explained

Lesson 16 Polymorphism in Action

Lesson 17 Comparing Objects

Lesson 18 Capstone 3

UNIT 4: MORE PROGRAMMING WITH OBJECTS

<i>Lesson 19</i>	<i>Pass by Value vs. Pass by Reference</i>
<i>Lesson 20</i>	<i>Garbage Collection</i>
<i>Lesson 21</i>	<i>Java Collections: List</i>
<i>Lesson 22</i>	<i>Java Collections: Set</i>
<i>Lesson 23</i>	<i>Collections: Queues</i>
<i>Lesson 24</i>	<i>Collections: Maps</i>
<i>Lesson 25</i>	<i>Using Generic classes</i>
<i>Lesson 26</i>	<i>Capstone 4</i>

UNIT 5: FILE PROCESSING

<i>Lesson 27</i>	<i>Reading from Files</i>
<i>Lesson 28</i>	<i>Writing to Files</i>
<i>Lesson 29</i>	<i>Using Streams from Java 8</i>
<i>Lesson 30</i>	<i>Data parsing</i>
<i>Lesson 31</i>	<i>File error handling</i>
<i>Lesson 32</i>	<i>Capstone 5</i>

UNIT 6: ADVANCED TOPICS

<i>Lesson 33</i>	<i>Exception Handling</i>
<i>Lesson 34</i>	<i>Lambda Functions</i>
<i>Lesson 35</i>	<i>Coupling/ Cohesion</i>
<i>Lesson 36</i>	<i>Design Patterns</i>
<i>Lesson 37</i>	<i>Singletons</i>
<i>Lesson 38</i>	<i>Encapsulation</i>

SPECIAL LESSONS:

<i>How to run Java programs from the terminal</i>
<i>How to use JShell</i>
<i>Package your app for deployment</i>

Unit 0

Getting Started with Java

1

Get Started

After reading lesson 1, you will be able to:

- **Install the NetBeans Integrated Development Environment (IDE)**
- **Configure IDE settings for writing Java programs**
- **Type in code, compile it, and run it using the IDE**

This lesson walks through the process of installing all the parts needed to write code and run your Java programs using the NetBeans IDE on both macOS and Windows. There has been much debate over whether the use of an IDE should be used to teach programming skills. The other option is to use a text editor to type your code, then use a command prompt window to compile and run the code. Personally, I prefer to start with an IDE for a few reasons:

- An IDE helps find syntax errors prior to code execution (avoids frustration)
- New programmers can build confidence in their ability to write code quickly
- Allows new programmers to concentrate on understanding the logic of coding, not the details of the language
- Provides tools to help debug when there are logic problems that are not readily seen in the code

When you first learn to program, using an IDE provides several advantages. For example, when typing your code using an IDE, there is instant feedback regarding syntax errors and even some logic errors, such as trying to use undeclared variables. Java is a strictly-typed programming language; every variable must be defined with a data type and in many cases initialized prior to use. Another nice feature of most IDEs is some form of code completion.

Consider this

You're comfortable programming in a different language than Java, and are familiar with basic object-oriented concepts. Now, you want to start programming in Java, the first thing you need to know is the difference in the syntax for Java compared to other languages. To begin with, one of the key syntactical differences is the use of semicolons in Java. Other languages, such as Python, use spacing and indentation to dictate blocks of code and the end of a statement. In Java, every statement must end with a semicolon. When typing code into an IDE, it provides an instant error and a hint if you forget to include the semicolon. Do you think using an IDE is an advantage when learning a new language?

1.1 What is an Integrated Development Environment

An Integrated Development Environment is designed to provide a one-stop shop. Rather than using a separate text editor to type your code, then invoke a compiler to create a class file (which is required for code execution), and finally running the program in a Java runtime environment, all three components are included in one software environment.

Another benefit of using the combined environment is the graphical user interface (GUI) that provides menus, toolbars, semantic coloring of the text as you type your code, immediate identification of syntax errors, code completion, and even some compile time errors can be identified when using an IDE.

One of my favorite parts of using an IDE is the debugging tool that is included. This tool can be used to find errors in your code, often referred to as 'bugs'. It can also be used to walk through the view your code execution one statement at a time and even view the values of all variables included in the program.

There are many different IDEs available for Java programming. Here are a few of the more common ones:

- NetBeans
- Eclipse
- IntelliJ
- BlueJ
- DrJava
- JDeveloper
- JCreator

Each IDE has its pros and cons, but for this book I have chosen to use NetBeans. NetBeans has many features to make coding more productive, such as:

- Automatically inserting matching braces, brackets, and quotes
- Code formatting
- Smart code completion
- Managing imports
- Ability to generate code

- Providing code templates for commonly used code snippets
- Helping to create the required code to generate a Javadoc
- Semantic code coloring

Did you know

In NetBeans, the code completion feature is usually turned on by default but it can be deselected by going to Tools/Options/Editor/Code Completion. If code completion is disabled, you can retrieve a list of code options by typing CTL-Space (Windows) or CMD-Space (mac).

Quick Check 1-1:

Which of the following is NOT a benefit of using an IDE:

- Combined text editor, compiler and runtime environment
- Code Completion
- Integrated debugging tool
- Automatic syntax error correction

1.2 Install the Java Development Kit (JDK)

Whether you are using an IDE or a text editor to write your code, first you must download the Java Development Kit, commonly referred to as the JDK. This is available from the Oracle website: <http://oracle.com>. When the JDK is downloaded, it automatically includes the Java Runtime Environment (JRE) and the Java APIs. The JDK provides the tools needed to write and compile Java code along with access to the prewritten code in the APIs. The JRE provides the runtime environment for deploying the programs and running them.

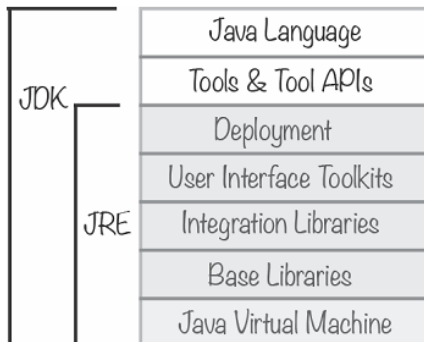


Figure 1.1 The JDK includes the Java Language, Tools, and the Java Runtime Environment (JRE)

It is important to choose the correct download based on your system; for example, I am using a Macbook Pro, so I chose the download for macOS. (Make sure to select the radio button to Accept License Agreement before starting the download.) If you are running windows and you have a 32-bit operating system, choose the x86 version of the downloads.

Java SE Development Kit 8u151		
You must accept the Oracle Binary Code License Agreement for Java SE to download this software.		
<input type="radio"/> Accept License Agreement <input type="radio"/> Decline License Agreement		
Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	77.9 MB	jdk-8u151-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	74.85 MB	jdk-8u151-linux-arm64-vfp-hflt.tar.gz
Linux x86	168.95 MB	jdk-8u151-linux-i586.rpm
Linux x86	183.73 MB	jdk-8u151-linux-i586.tar.gz
Linux x64	166.1 MB	jdk-8u151-linux-x64.rpm
Linux x64	180.95 MB	jdk-8u151-linux-x64.tar.gz
macOS	247.06 MB	jdk-8u151-macosx-x64.dmg
Solaris SPARC 64-bit	140.06 MB	jdk-8u151-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	99.32 MB	jdk-8u151-solaris-sparcv9.tar.gz
Solaris x64	140.65 MB	jdk-8u151-solaris-x64.tar.Z
Solaris x64	97 MB	jdk-8u151-solaris-x64.tar.gz
Windows x86	198.04 MB	jdk-8u151-windows-i586.exe
Windows x64	205.95 MB	jdk-8u151-windows-x64.exe

Figure 1.2 Screenshot from Oracle website with information on JDK downloads

Once the download is complete, run through the installation process by double clicking on the download and following the installation instructions.

Quick Check 1-2:

The JDK includes:

- a. Java Language, Tools, and JRE
- b. Java Language and a debugger
- c. Tools, debugger, and JVM

1.3 How to Install NetBeans

There are two options for downloading the NetBeans IDE. The Oracle website can be used to download the JDK alone or there is an option to download NetBeans with the JDK. This saves you the extra step of downloading two separate files.

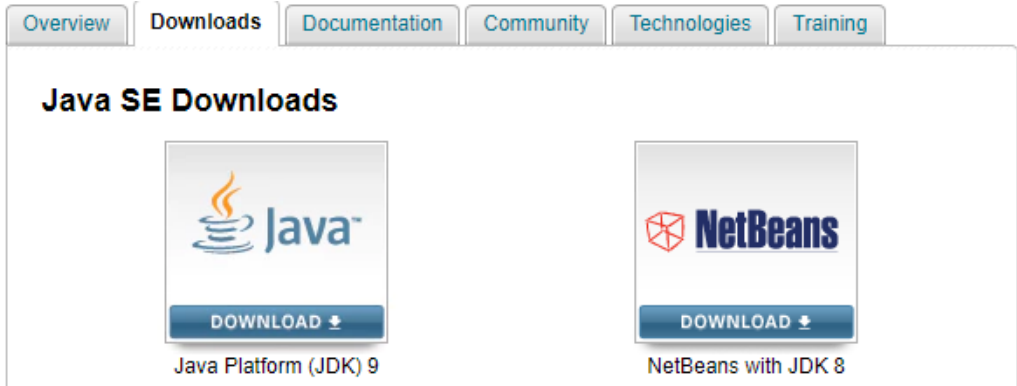


Figure 1.3 Screenshot of Installation options for standalone JDK or NetBeans with JDK

If you already installed the JDK, then it is not necessary to download it again and you can use the NetBeans website to download the latest version of NetBeans (<https://netbeans.org/downloads/>).

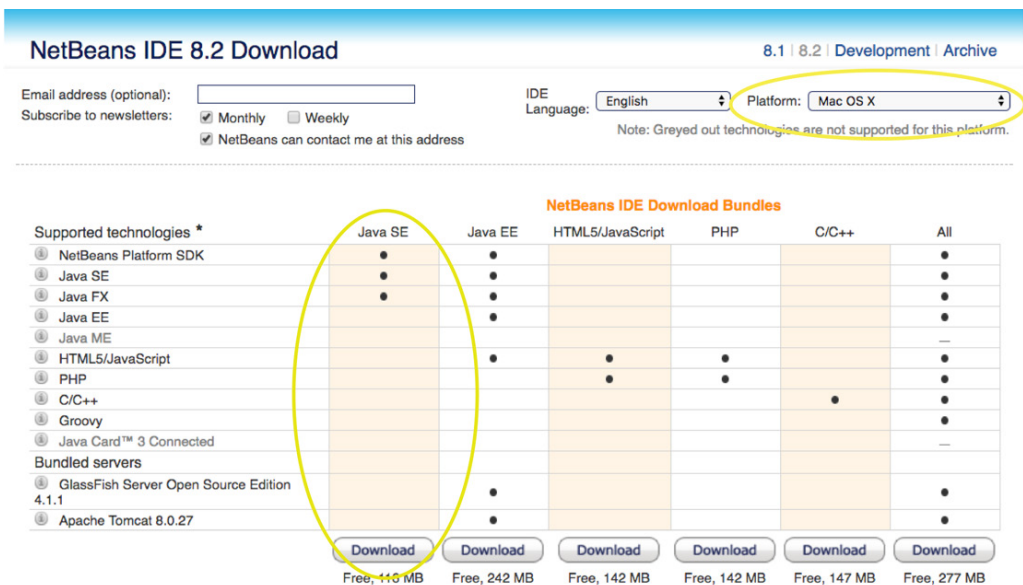


Figure 1.4 Screenshot of the NetBeans IDE Download page showing all download options

There are several options to choose from, but to follow along with the examples in this book, choose Java SE download. Also notice the top of the window and make sure you have chosen

the correct platform. In Figure 1.4, my platform is Mac OS X. The NetBeans website has a link to Installation Instructions. It is a good idea to check that link since you might be working with a newer version of the product.

Once the download is complete, launch NetBeans. In the next section, we will review some of the environment settings to check before continuing with our activities. Figure 1.4 shows a screenshot of the installation process on a Mac. As you can see, the installation wizard makes it easy to install. Start by reading the **Introduction**, then click the **Continue** button (see Figure 1.5). The next screen provides all the license information. You must click the **Agree to** button. The **Destination Select** gives you the option to change the default location, then click the **Continue** button again. At the next window, click the **Continue** button. At the very end, you will get a summary of the install.

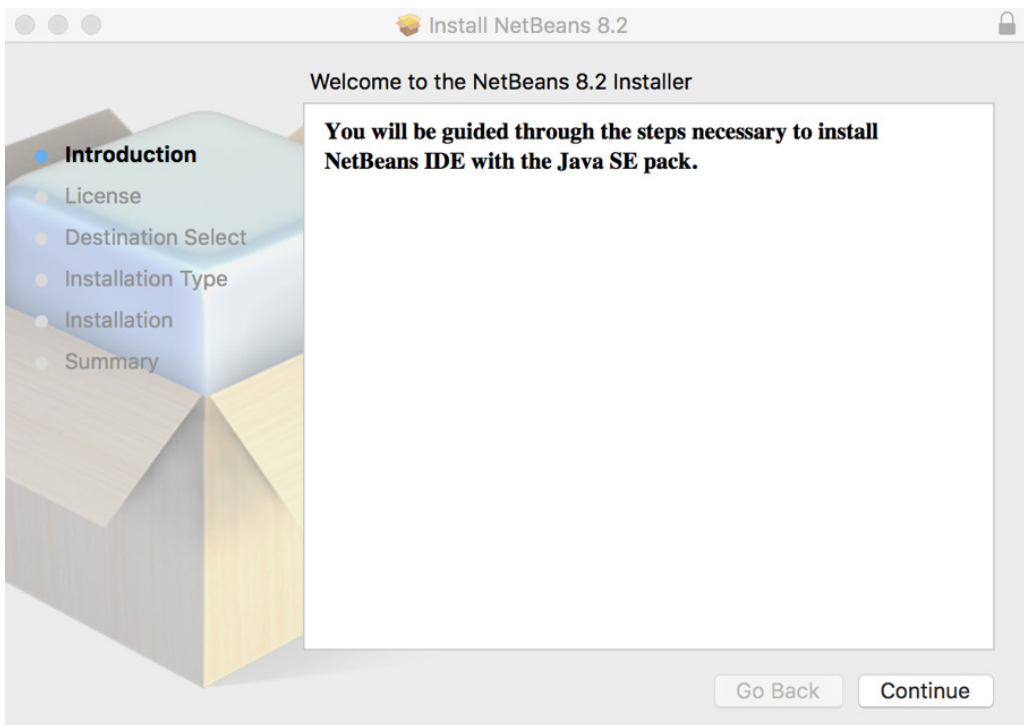


Figure 1.5 Screenshot of NetBeans Installation on a macOS

When NetBeans is opened for the first time, you will see the Start Page (see figure 1.6) From the Start Page, take a look at the links available under the Demos & Tutorials or click the button 'Take a Tour' to learn about NetBeans.

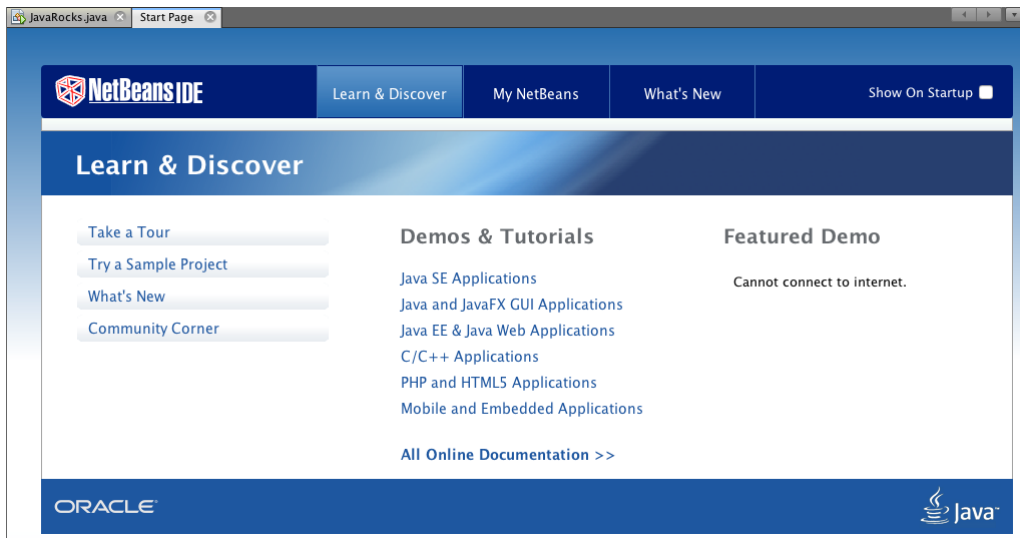


Figure 1.6 Screenshot of the Start Page in NetBeans

Did you know

Every time you open NetBeans, the Start Page will automatically show up. On the upper right corner of the Start Page, you can deselect the 'Show on Startup' checkbox. If you accidentally changed the checkbox but you want to see the Start Page, simply go to the Help option and choose Start Page.

Quick Check 1-3:

True or False: If I accidentally closed the 'Start Page', I can't get back to the page without restarting NetBeans.

1.4 NetBeans Environment Setup

The first setting that you want to check in your NetBeans installation is the Java Platform version. Go to Tools/Java Platforms, make sure that the Platform name and Platform Directory are set to your version of the JDK. On a Windows machine, the JDK is usually in this folder: C:\Program Files\Java\ by default. For a macOS, the JDK is located here: /Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home.

Next, depending on what other programming languages and IDEs you might have used, you might want to personalize the settings such as the background color, the font size, and so on. The default profile is called NetBeans and it provides a white background. There are several themes that are included in the download. On a Windows machine, you can go to Tools/Options/Fonts & Colors. For a mac, go to NetBeans/Preferences/Fonts & Colors. Many

programmers prefer a black background, so feel free to change it to your preference. Figure 1.5 show the NetBeans Options window on a Mac.

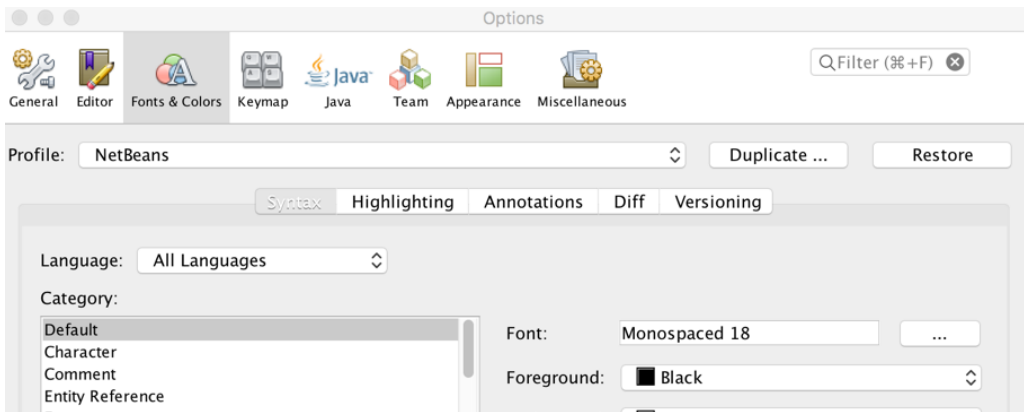


Figure 1.7 Screenshot of the Options menu in NetBeans

Quick Check 1-4:

To change the look and feel of the IDE code window, change the _____:

- a. Keymap
- b. Profile
- c. Category
- d. Appearance

1.5 Write a Java Program Using NetBeans

Now that we have installed all of the parts needed to start programming in Java, let's write our first program.

In the NetBeans IDE, start by using the File menu, choose New Project. A dialog box appears, choose Java /Java Application and hit Next.

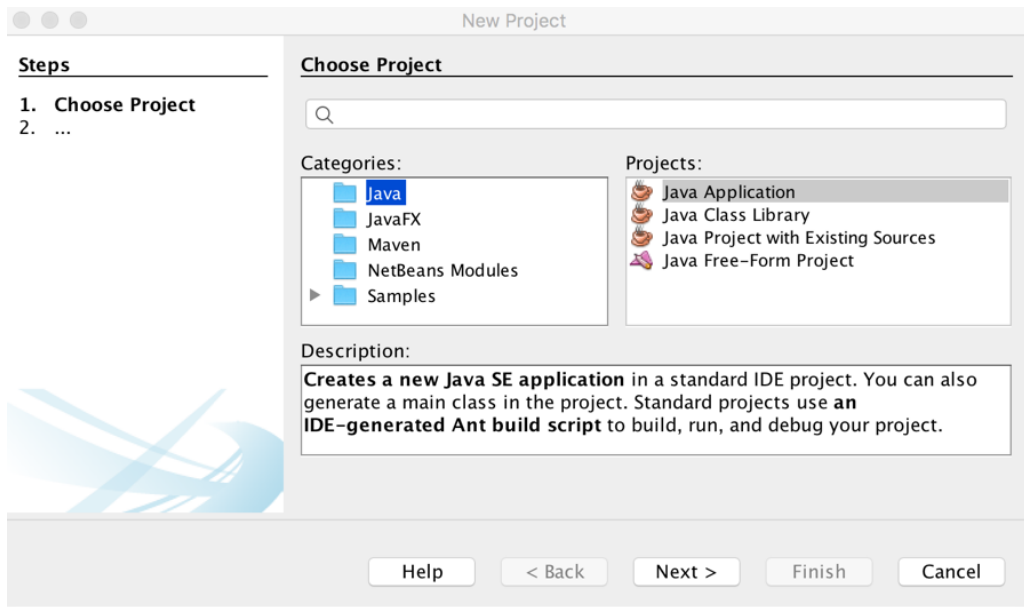


Figure 1.8 Screenshot of first step to create a new project in NetBeans

Give the application a name; for our example, name the project "JavaRocks", then click Finish.

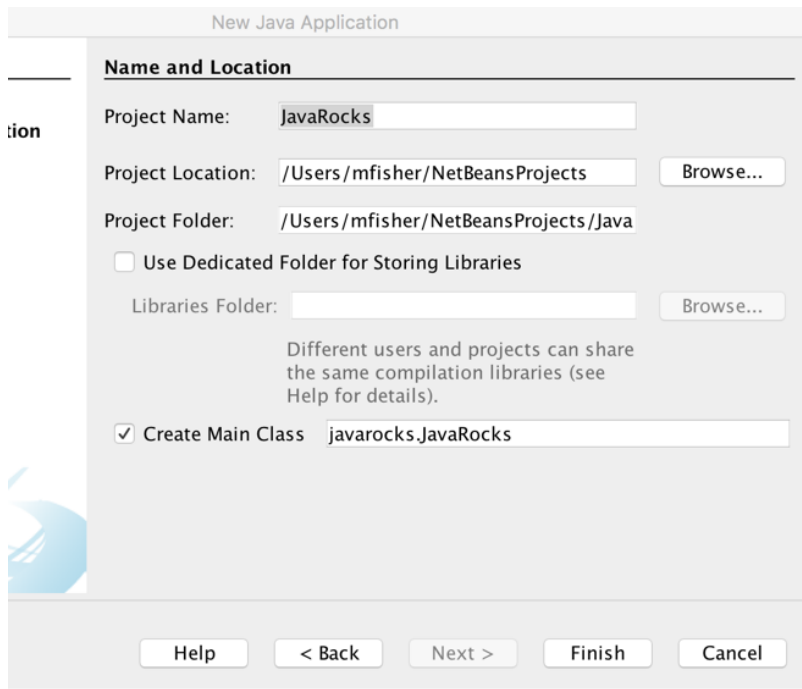


Figure 1.9 Screenshot of NetBeans, here we give our project a name

The IDE creates a project folder with several subfolders and a file with a .java extension. This is where we write our Java code. Figure 1.8 shows the program that gets created in NetBeans.

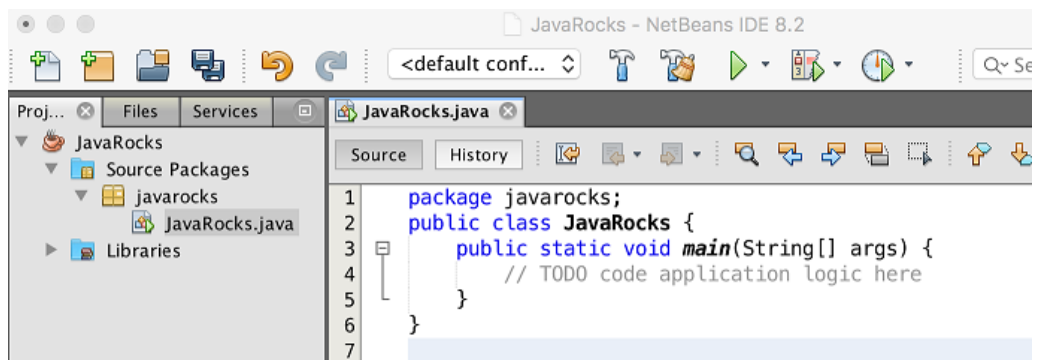


Figure 1.10 A screenshot of our first Java program

Every Java project must include at least one file that contains a main method. In our first program, line 3 is the start of this method. The main method extends from line 3 where it starts with an open curly brace and ends on line 5 with a closing curly brace. In the next few lessons, we will talk more about exactly what each line of code is doing in this example.

Before we run this new program, let's add a line of code so we can see it print information to the console. The console is also part of the IDE. Keeping it simple, replace the comment on line 4 that currently says `// TODO code application logic here` with this line of code:

```
System.out.println("Java Rocks!!");
```

Make sure that you type the line of code exactly as it appears above. Java is case-sensitive, so you must use a capital S for System and lower case for the remaining command. The words inside the double quotes can be mixed case - they are considered a literal constant. Next, we want to save the program, make sure that you don't change the file name. In Java, the file name must match the class name exactly. Once you are done, click the green arrow to the right of the hammer and brush. You should see output similar to Figure 1.9.

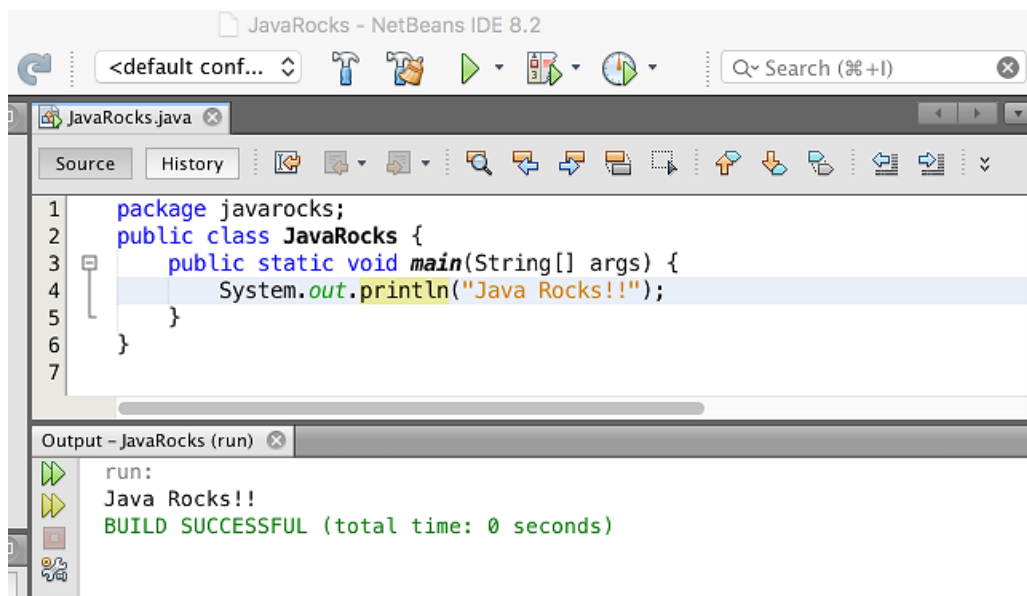


Figure 1.11 Screenshot of the output from running the program JavaRocks

Quick Check 1-5:

Every Java project must have at least one _____:

- a) comment
- b) `println` statement
- c) `main` method

1.6 Summary

In this lesson, we reviewed the installations necessary to write Java code, compile your code, and run your code. We also reviewed instructions for downloading an Integrated Development Environment.

In the next lesson, we will work on writing code for a program that acts like a calculator. This lesson is being used as an introduction to the syntax of Java and Java libraries. Some programming languages, such as Python, do not require as much explicit notation such as curly brackets to indicate blocks of code. Instead, Python stresses lines and indentation instead of explicit curly brackets used in Java.

TRY THIS: Once you have the Java JDK downloaded and the NetBeans IDE installed, create a new project. Use the sample code from this lesson and instead of printing "Java Rocks", print your name.

QUICK CHECK 1-1:

Which of the following is **NOT** a benefit of using an IDE:

- a. Combined text editor, compiler and runtime environment
- b. Code Completion
- c. Integrated debugging tool
- d. **Automatic syntax error correction**

QUICK CHECK 1-2:

The JDK includes:

- a. **Java Language, Tools, and JRE**
- b. Java Language and a debugger
- c. Tools, debugger, and JVM

QUICK CHECK 1-3:

- a. **False:** If I accidentally closed the 'Start Page', I can't get back to the page without restarting NetBeans. **To find the 'Start Page', click 'Help' in the menu bar and choose 'Start Page'.**

QUICK CHECK 1-4:

To change the look and feel of the IDE code window, change the _____:

- a. Keymap
- b. Profile
- c. Category
- d. Appearance

QUICK CHECK 1-5:

Every Java project must have at least one _____:

- a. comment
- b. println statement
- c. main method

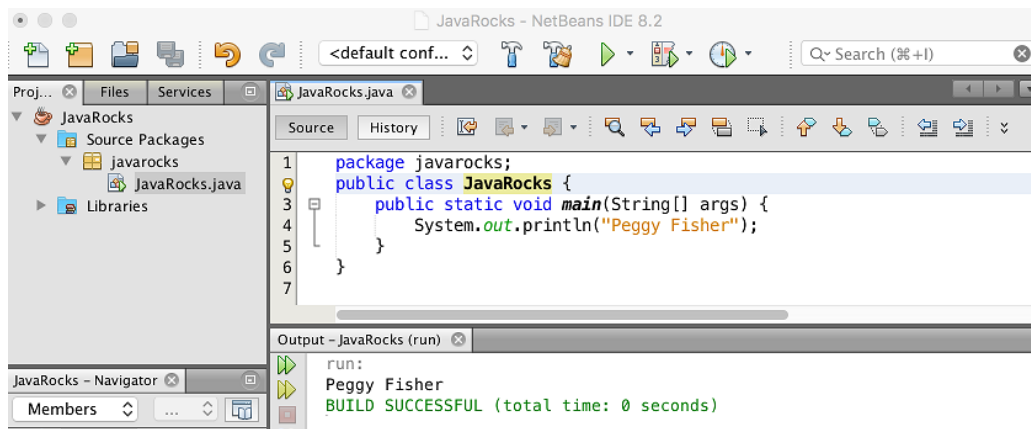
ANSWER TO THE TRY THIS:

Figure 1.12 Screenshot from NetBeans for the updated code and the output message

Listing 1.1: Java program that prints out your name

```

1 package javarocks;
2 public class JavaRocks {
3     public static void main(String[] args) {
4         System.out.println("Peggy Fisher"); // #A
5     }
6 }

```

#A In the println statement, I changed "Java Rocks" to my name, "Peggy Fisher"

Unit 1

Classes and Objects

2

Creating Classes

After reading lesson 2, you will be able to:

- Define a new class
- Add fields to represent the class attributes
- Add methods to the class to represent class behaviors
- Create objects based on the class definition

Java is an object-oriented programming language, but in order to create objects we have to start by creating a class. A **class** in Java is a blueprint or template for an object. So, what does that mean? Java programs create objects that model items in the real world. These items can represent physical items such as a car or more abstract objects such as a bank account. Before we can create a class, we must identify the attribute(s) and behavior(s) of our objects. I find it helpful to think of the attributes as the adjectives used to describe the object and the behaviors are the verbs or actions that we can perform on the object. For example, a car has the following attributes:

- Make
- Model
- Year
- MPG
- Color

And the behaviors include:

- Move forward
- Move backward
- Stop (or brake)
- Turn left/right
- Honk horn

Consider This

You have been hired by a local car dealership to provide their IT support. The first project is to create a mailing list for all their customers. The owner wants to automate the process of sending out reminders for customers that need maintenance appointments for their vehicles. In order to write a program to automate this process, what class(es) do you think are needed for this project?

2.1 Defining a Class

Remember, a class is like a blueprint (or template) for the objects needed in your program. The best way to understand how to create a class is to walk through an example. So, I am going to create a class that represents a car. To understand the type of information we need about each car, table 2.1 contains sample information about three cars.

Table 2.1: Attributes for three cars

make	model	year	mpg	color
Subaru	Outback	2017	28	black
Chevrolet	Malibu	2015	30	grey
Ford	Fusion	2018	32	blue

It is time to create our car class in Java. It is possible to use an IDE to create our class, but since this is our first program, I am going to just use a text editor. To start, I create a file named Car.java.

Note about Java naming standards

For this example, the Car class is created in a file named Car.java. The Java programming language **requires** the file name to match the class name exactly including mixed case.

Inside the text file, start by using the keyword **class** followed by the class name. It is recommended that all class names start with an upper-case letter. Next, I have added all of the attributes identified for a car using the appropriate data types and variable names. For the variables that contain descriptive words, I have identified them as `String` variables. Since the year is numeric without any decimal points, I declared it as an integer, which in Java is `int`. And finally, the field for MPG or Miles Per Gallon is a `double` since it is a floating-point number.

```

1 class Car {
2     String make;
3     String model;
4     int year;
5     double mpg;
6     String color;
7
8 }

```

Figure 2.1 Code listing for the start of the Car class

Note:

The String data type is actually a class, therefore it must be coded with an upper-case S, as opposed to the lower-case letters for int and double. The String class is discussed in more detail later in this book.

The data fields for a class are often referred to as instance data. It is called instance data since the values are specific to an instance of the class and are not shared among all instances of the same class. Instance variables (fields) are defined directly inside the class.

In table 2.1, the first row represents a single object, which is called an instance of the car class. This specific object has instance data detailing that this car make is a Subaru, model is a Outback, year is 2017, mpg is 28 and the color is black. This data is not shared with the other two rows in our table. Each car has its own information.

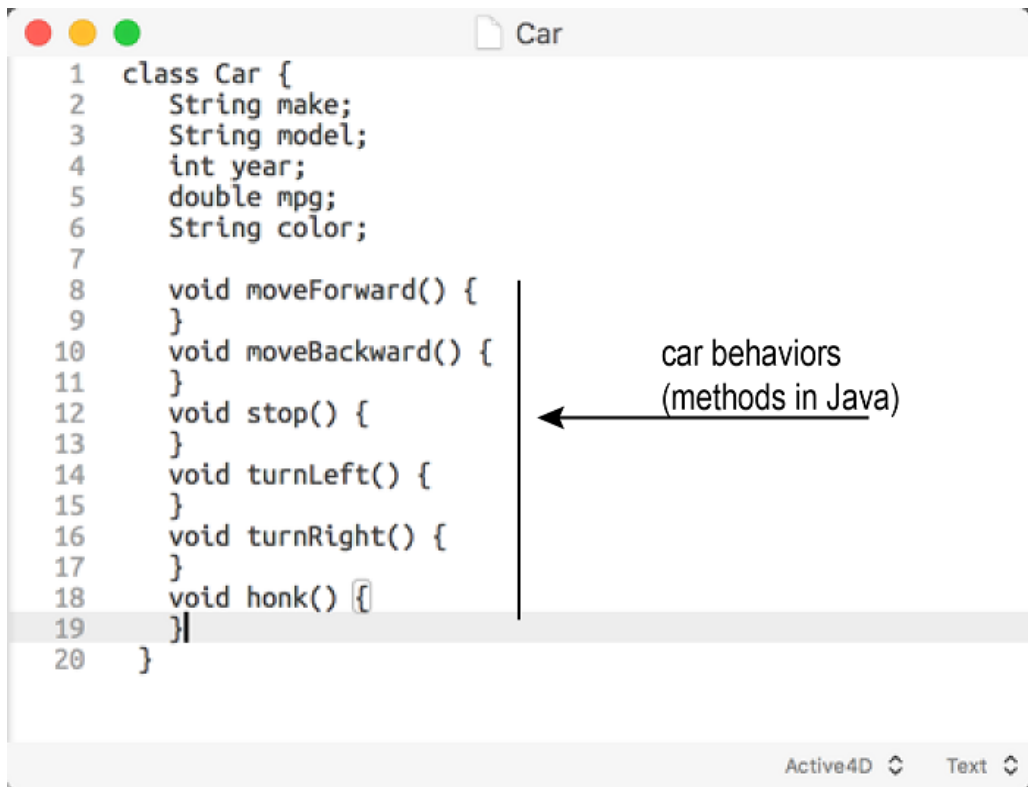
Quick Check 2.1, fill in the blank and multiple choice:

1. A class is the _____ for an object.
2. A class contains the _____ and _____ for an object, choose one:
 - a. attributes, behavior
 - b. name, data type
 - c. child, parent
3. The class defines the _____ data for an object.
4. Given the class name: MyFirstClass, what is a valid file name for this class:
 - a. myfirstclass.java
 - b. MyFirstClass.java
 - c. anyname.java

2.2 Add Class Behaviors using Methods

Now that I have added the Car class and the fields for that class, the next important step is adding the behaviors to our class. Each behavior is added as a method in Java. A detailed explanation of methods in Java is in a later lesson, but for this example, I am adding a few simple methods.

In figure 2.2, I have added methods that represent the behaviors of a car. Each method has the keyword `void` before the method name. This keyword indicates that the method does not return any values.



```

1  class Car {
2      String make;
3      String model;
4      int year;
5      double mpg;
6      String color;
7
8      void moveForward() {
9      }
10     void moveBackward() {
11     }
12     void stop() {
13     }
14     void turnLeft() {
15     }
16     void turnRight() {
17     }
18     void honk() {
19     }
20 }

```

car behaviors
(methods in Java) ←

Figure 2.2 Code listing of the Car class with methods added

The methods that are declared inside of a class can be used once we create an instance of the class. In other words, once we have an object defined. In the next section I will show you how to use the Car class to create car objects.

Quick Check 2.2:

Which statement is not a behavior for the Car class:

- a. moveForward()
- b. setColor()
- c. honk()

2.3 Create objects based on the class definition

Now that we have created our Car class, the next step is to create objects that hold the instance data about each car.

In Java, each object is given a variable name and then the keyword 'new' is used to create the object. The process of creating an object from a class is called **instantiation**. In other words, we are creating a single instance of an object using the class as the model. For this activity, I will create a new car object. In Listing 2.2, starting on line 20 I added a main method to the program.

Listing 2.2: Creating a new car object

```

1.  class Car {
2.      String make;
3.      String model;
4.      int year;
5.      double mpg;
6.      String color;
7.
8.      void moveForward() {
9.      }
10.     void moveBackward() {
11.     }
12.     void stop() {
13.     }
14.     void turnLeft() {
15.     }
16.     void turnRight() {
17.     }
18.     void honk() {
19.     }
20.     public static void main(String[] args) { // #A
21.         Car myCar = new Car(); // #B
22.         myCar.make = "Subaru"; // #C
23.         myCar.model = "Outback"; // #C
24.         myCar.year = 2017; // #C
25.         myCar.mpg = 28; // #C
26.         myCar.color = "black"; // #C
27.     }
28. }
```

#A Every Java program requires a starting point defined as a main method

#B This statement instantiates a new Car object named myCar

#C Using dot notation, the instance data for the myCar object is populated with real values

Note:

When executing a Java program, the Java Virtual Machine (JVM) searches the code for the main method: `public static void main(String[] args)`. This is a requirement in every Java application. This is the starting point of execution in Java.

Inside the main method, line 21 creates the first car object using the Car class. Figure 2.3 explains the parts of this statement.

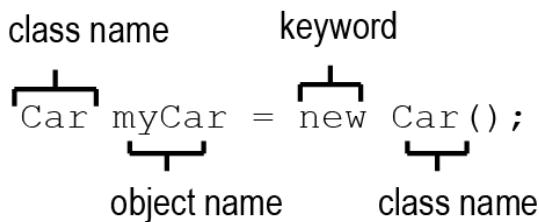


Figure 2.3 Diagram of the statement used to create a new car object from the Car class

This statement creates the new object `myCar`. Once the object is created, the next step is to populate the data fields. Lines 22-26 assign values to the data fields in `myCar`.

In each of these assignment statements, I use dot notation to access the fields for the object `myCar`. In this example, the dot notation identifies the object followed by a dot and then the variable name of the instance data. In our example, we have `myCar.make = "Subaru";`. This updates the car make for the object called `myCar`.

In the next Lesson, I will review some best practices when using object-oriented programming to create objects and retrieve/modify the instance data using methods.

Quick Check 2.3:

1. What keyword is used when creating an object for the first time:
 - a. new
 - b. this
 - c. that

2. The process of creating an object from a class is called:
 - a. new
 - b. instantiation
 - c. encapsulation

2.4 Summary

In this lesson, you learned:

- Define a new class
- Add variables to represent the class attributes
- Add methods to the class to represent class behaviors
- Create objects based on the class definition

Creating classes and objects is an essential part of any Java program. Remember: the class is the blueprint used to create the template or model of our real-world objects. Then, we use the class to create one or more objects.

In the next lesson, I will add more methods to the class that can be used to retrieve and modify instance data for an object.

Try this:

A friend owns a pet grooming business. For this activity, we want to create a class in Java to represent a pet. Each pet must have a name, pet type, owner, and age.

Then write a main method that creates at least three pet objects and provides values for the instance data of each object.

Quick Check 2.1 Solution:

1. A class is the blue print for an object.
2. A class contains the _____ and _____ for an object, choose one:
 - a. **attributes, behavior**
 - b. name, data type
 - c. child, parent
3. The class defines the instance data for an object.
4. Given the class name: MyFirstClass, what is a valid file name for this class:
 - a. myfirstclass.java
 - b. **MyFirstClass.java**
 - c. anyname.java

Quick Check 2.2 Solution:

2. Which statement is not a behavior for the Car class:
 - a. moveForward()
 - b. **setColor()**
 - c. honk()

Quick Check 2.3 Solution:

1. What keyword is used when creating an object for the first time:
 - a. **new**
 - b. this
 - c. that

2. The process of creating an object from a class is called:
 - a. new
 - b. **Instantiation**
 - c. encapsulation

Solution to Try This Exercise:

It is important to remember that everyone programs slightly different, so your solution might not be identical to mine, but that is o.k.

Listing 2.3 This is a sample Pet class used to represent a pet

```
class Pet {                                     //#A
    String petName;                             //#B
    String petType;                             //#B
    String owner;                               //#B
    int petAge;                                 //#B

    public static void main(String[] args) {    //#C
        Pet p1 = new Pet();                    //#D
        p1.petName = "Harley";                 //#E
        p1.petType = "dog";                    //#E
        p1.owner = "Cy Fisher",                //#E
        p1.petAge = 4;                          //#E

        Pet p2 = new Pet();                    //#D
        p2.petName = "Harley";
        p2.petType = "dog";
        p2.owner = "Cy Fisher",
        p2.petAge = 4;

        Pet p3 = new Pet();                    //#D
        p3.petName = "Harley";
        p3.petType = "dog";
        p3.owner = "Cy Fisher",
        p3.petAge = 4;

    }
}
```

#A Declare the start of the Pet class

#B Instance data for each pet includes name, type of pet, owner's name and pet age

#C All Java applications require a main method, used as the starting point for execution

#D Instantiate a new pet object

#E Use dot notation to provide values for the instance data

3

Visibility Modifiers

After reading lesson 3, you will be able to:

- Organize your application using packages
- Understand the options for visibility modifiers
- Choose the correct modifier for your code

The visibility modifiers determine the access level of the instance data, method, class, etc. In this lesson I will explain the concept of packages in Java, which is necessary to understand how visibility modifiers affect your application and review the various types that can be used.

One reason for using these modifiers is to restrict access. Think about a banking application. One of the classes would probably be for a bank account. The bank account class might have a customer's name, home address, birth date, etc. If access to these fields is marked as public, then there is no way to validate that changes are made correctly. For example, the birth year could be set to an invalid year such as 2100.

Consider This

The human resources department for Company A uses a Java application to keep track of all employees. The application has an Employee class that contains all the demographic and pay information about the employee. Only employees in HR have access to the sensitive information such as an employee's full name, social security number, home address, pay scale, etc. What would happen if all of the instance data for the Employee class was marked as public? How can we protect this data from getting updated with invalid information?

3.1 Packages

In Java, a package is a namespace used to group a set of classes together. So far, our application consisted of only one file with one class, but most Java applications consist of many files and many classes. The package allows us to connect these classes together.

When using a package, the package name appears at the very top of your code inside your class file. Java naming standards recommend that the package name is written in all lower-case letters to avoid confusion with any class names. To add a class to a package, use the keyword `package` followed by the package name, for example:

```
package packagename;
```

As we continue to create more complicated classes, one of the benefits of Java is the vast amount of code already written and tested that is available for general use. These packages are included in the Java API (Application Programming Interface). The API is a library of packages that can be used in our applications. Most of these packages are designed for handling common programming requirements, such as working with String objects or File objects. This allows us to concentrate on the programming needed to handle the business requirements for our program.

To leverage other classes in Java that are not part of your current package, we can use the `import` statement. The `import` statement allows us to 'import' other packages that contain one or more classes. For example, any program that reads information from the console uses the `Scanner` package and must include this statement: `import java.util.Scanner;`

Quick Check 3.1:

Packages are used for:

- a. grouping multiple classes together
- b. mailing a care package to a loved one
- c. adding graphics to our program

3.2 Visibility Modifiers Explained

There are four possible values used to control access to your code: `public`, `protected`, default (this is the value when the modifier is omitted), and `private`. Table 3.1 shows each type of modifier, as you can see it goes from least restrictive (`public`) to most restrictive (`private`).

Table 3.1: Access Modifiers in Java

Modifier	Class	Package	Subclass	Everywhere
<code>public</code>	Y	Y	Y	Y
<code>protected</code>	Y	Y	Y	N
<code>none specified</code>	Y	Y	N	N
<code>private</code>	Y	N	N	N

It is important to always choose the modifier that is most restrictive but still allows access when needed. Let's take a closer look at each type:

- **public** – this is the least restrictive and allows access to this code from the current class, the package, any subclasses, and any other classes.
- **protected** – this modifier allows the same access as public except that it can only be accessed by the current class, any classes in the same package, and any subclasses.
- **default** – when the modifier is omitted, it is considered a default visibility. This restricts access to only the class and classes in the same package
- **private** – this is the most restrictive and only allows access within the same class

Lesson 2 introduced the topic of classes and objects which are required for any object-oriented (OO) programming language. In a later lesson, I will review the concept of Inheritance in much more detail, but to help understand the concepts of visibility modifiers, I want to explain the relationship of a class and subclass. In Java, we can create a 'child' class using another class as the 'parent'. This new class becomes a subclass and the parent is called the superclass. This is one of the strengths of Java, it is referred to as inheritance. Inheritance allows us to create a class that inherits all the public and protected information from the superclass and adds any additional information necessary for the subclass. For example, our Car class could be a subclass of a Vehicle class.

Let's look at a diagram that depicts two packages. Both packages have two classes, but package B has a class that is actually a subclass of the Vehicle class in package A.

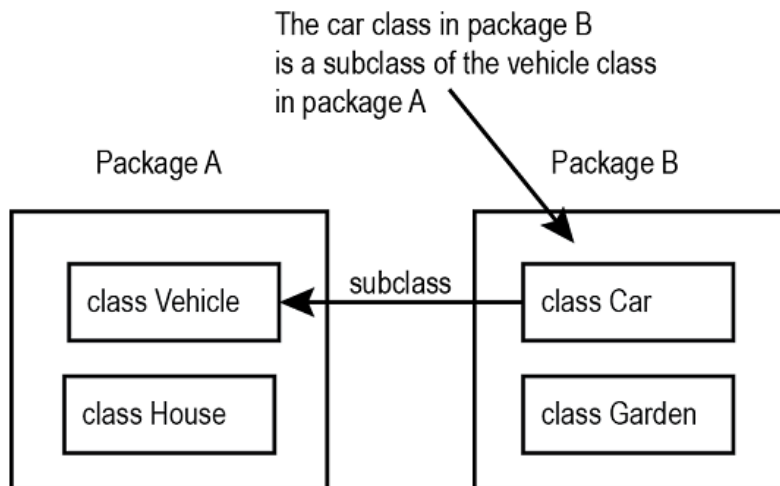


Figure 3.1 Diagram of two packages to demonstrate the impact of the visibility modifiers

In this example, Car is a subclass of Vehicle. Package A has two classes: Vehicle and House. Package B has two classes: Car and Garden. Table 3.2 depicts the impact of using each of the visibility modifiers with these two packages. The table shows the visibility based on the modifier in the Vehicle class.

For example, if the Vehicle class has a data field marked protected, then only the classes in the same package (Vehicle and House) and the subclass (Car) can access that data. Note, when the data is marked as private in the vehicle class, it is not directly accessible by any of the other classes.

Table 3.2 Access to the other classes from the Vehicle class

Modifier	Vehicle	House	Car	Garden
public	Y	Y	Y	Y
protected	Y	Y	Y	N
none specified	Y	Y	N	N
private	Y	N	N	N

Quick Check 3.2:

1. Which of the following is not a visibility modifier in Java:
 - a. public
 - b. private
 - c. package
2. T/F: When the private modifier is used, only a subclass access that data.
3. Which of the following is true about a subclass:
 - a. A subclass is a duplicate of the superclass
 - b. A subclass can access all public and private information from the superclass
 - c. A subclass can access public and protected data from the superclass

3.3 Visibility Modifiers in Action

The next step is to use this information to update our Car class with the appropriate visibility modifiers. Listing 3.1 shows that the updated Car class, specifically all data is marked as private and methods are public. The next lesson details the impact of using these modifiers and how it helps with data integrity.

Listing 3.1: Car class updated with visibility modifiers

```
1. public class Car {           // #A
2.     private String make;    // #B
```

```

3.     private String model;  //#B
   private int year;        //#B
4.     private double mpg;    //#B
5.     private String color;  //#B
6.
7.     public void moveForward() {  //#C
8.     }
9.     public void moveBackward() {  //#C
10.    }
11.    public void stop() {          //#C
12.    }
13.    public void turnLeft() {      //#C
14.    }
15.    public void turnRight() {     //#C
16.    }
17.    public void honk() {          //#C
18.    }
19.    public static void main(String[] args) {
20.    Car myCar = new Car();
21.        myCar.make = "Subaru";
22.        myCar.model = "Outback";
23.        myCar.year = 2017;
24.        myCar.mpg = 28;
25.        myCar.color = "black";
26.    }
27.    }

```

#A The Car class is now declared as public

#B The instance data is marked as private, only methods in the Car class can directly update this data

#C All methods are public, making them available everywhere

In this example, all the instance data have been updated and marked as private. This is used to make sure that the data is not updated inadvertently. In OO terms, this allows all the Car information to be encapsulated in the Car class. Each of the methods in this class have been updated as public. That way, other programs can use the methods once they create a Car object, such as `myCar.honk()`;

3.4 Summary

In this lesson, you learned:

- How to organize your application using packages
- What visibility modifiers are in Java
- Choosing the correct modifier in your code

Remember to choose the most restrictive modifier that still provides the access required. In Java, there are four types of modifiers: public, protected, default and private.

In the next lesson, I will add more methods to a class and show you how to put the class in a separate file.

Try this:

A friend owns a pet grooming business. Using the class you created from Lesson 2, add the visibility methods to the class, instance data and methods.

Quick Check 3.1 Solution:

1. Packages are used for:
 - a. **grouping multiple classes together**
 - b. mailing a care package to a loved one
 - c. adding graphics to our program

Quick Check 3.2 Solution:

1. Which of the following is not a visibility modifier in Java:
 - a. public
 - b. private
 - c. **package**
2. T/F: When the private modifier is used, only a subclass access that data. False
3. Which of the following is true about a subclass:
 - a. A subclass is a duplicate of the superclass
 - b. A subclass can access all public and private information from the superclass
 - c. **A subclass can access public and protected data from the superclass**

Solution to Try This Exercise:

It is important to remember that everyone programs slightly different, so your solution might not be identical to mine, but that is o.k.

Listing 3.2 The Pet class updated with visibility modifiers

```
public class Pet {                                //#A
    private String petName;                       //#B
    private String petType;                       //#B
    private String owner;                         //#B
    private int petAge;                           //#B

    public static void main(String[] args) {
        Pet p1 = new Pet();
        p1.petName = "Harley";                    //#C
        p1.petType = "dog";
        p1.owner = "Cy Fisher",
        p1.petAge = 4;

        Pet p2 = new Pet();
        p2.petName = "Harley";
        p2.petType = "dog";
        p2.owner = "Cy Fisher",
```

```
p2.petAge = 4;

Pet p3 = new Pet();
p3.petName = "Harley";
p3.petType = "dog";
p3.owner = "Cy Fisher",
p3.petAge = 4;

}
}
```

#A The Pet class is now declared as public

#B The instance data is marked as private, only methods in the Pet class can directly update this data

#C Since these statements are in the same class, the main method can update the instance data

4

Adding Methods

After reading lesson 4, you will be able to:

- Understand the components of a method signature
- Add Getter and Setter methods to your class
- Add more methods to your program

All programming languages have a way to create a block of reusable code. In Python and other programming languages it is called a function, in Java it is called a method. The idea is that the method performs some action, but the calling program does not need to know exactly how it works, just that it will return the correct result or perform the correct action.

In Lesson 3, I introduced the topic of encapsulation as a pillar of OO programming. A key feature of encapsulation is using methods to access the private data of a class. The methods can contain logic to prevent any erroneous data corruption, such as trying to withdraw an amount more than the current balance of a checking account.

Consider This

Have you ever made microwave popcorn? Some microwaves have a button with a label that says 'popcorn'. When you place the bag of un-popped popcorn in the microwave and push that button, it starts to heat up the bag until the popcorn kernels get hot enough to pop. After a certain amount of time, it shuts off and you have a bag of popped popcorn. This is similar to a method in Java. The microwave has a 'popcorn' method.

Can you think of other examples related to common household objects that might be considered an object with one or more methods?

4.1 Method Signatures

In Java, a method signature paints a picture about how to use the method. The only required elements of a method signature are the method's return type, the method name, parentheses

for the parameter list (which can be empty), and the method body inside curly braces (which can also be empty).

In addition to these required elements, methods often include a visibility modifier, and a parameter list inside the parentheses. The keyword `static` is included when the method is not associated with a particular object, like the main method. Static methods are addressed later in the book. Figure 4.1 describes each part of a method signature, including required and optional components.

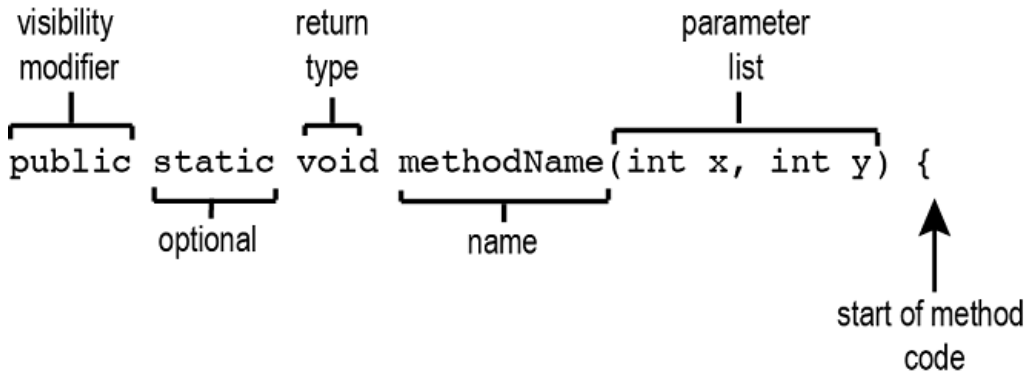


Figure 4.1 Diagram of the components in a method signature

4.1.1 Visibility Modifier

In lesson 3, I reviewed visibility modifiers in detail, now let's see how to use them in our methods. Remember, a method can have the following visibility modifiers: `public`, `protected`, default and `private`. It is also possible to omit the visibility modifier in which case the method is only visible to other classes that share the same package name. Table 4.1 details the options for visibility modifiers and their impact on the code.

Table 4.1: Visibility Modifiers in Java

Modifier	Class	Package	Subclass	Everywhere
<code>public</code>	Y	Y	Y	Y
<code>protected</code>	Y	Y	Y	N
default	Y	Y	N	N
<code>private</code>	Y	N	N	N

The visibility modifier plays a key role in enforcing data integrity, so choose your modifier carefully. Try to use the strictest modifier that still provides the access you need.

4.1.2 Return Type

The return type identifies what type of information is returned from the method. The return type can be a number, a String, a character, and so on. But, there are times when a method does not need to return any values to the calling program. In this case, we use the keyword **void** to indicate that the method does not return anything. It might be a method that takes information and prints it to the console. In this case there is nothing to return.

It is important to point out that a Java method is restricted to returning only one type of data. For example, a method cannot return both a String and a double value, it has to return either a String or a number defined as a double. To get around this limitation, it is possible to return an object and I will address that in a later lesson.

Table 4.2 has several method signatures and example return values:

Table 4.2: Sample method signatures

Method Signature	Return type	Sample return values
<code>public void popcorn()</code>	void	Does not return any values
<code>public String getFirstName()</code>	String	"Peggy"
<code>public double averageQuizScore()</code>	double	98.74
<code>public int countEmployees()</code>	int	125
<code>public char middleInitial()</code>	char	'J'
<code>public Car getCar()</code>	car	Car object, which contains: "Chevrolet", "Camaro", 2018, 24.5, "Navy"

4.1.3 Method Naming Conventions

Most programmers use camel case when naming the methods in Java, such as `findAverage`. The first letter is in lowercase and the first letter of each subsequent word is capitalized. Try to use meaningful names that describe the function of the method. For example, a payroll application might have a method named `calculateEarnings`.

4.1.4 Parameter List

All methods are **required** to have a parameter list, even if it is empty. The parentheses in the method signature identify it as a method. The parameter list provides information to the method from the calling program. For example, if we need to change the last name of an employee, the new last name can be included in the parameter list. Then the method is used

to change the instance data field for the employee last name. Here is the method signature for this method:

```
public void setEmployeeLastName(String lastName)
```

The parameter list in the method signature identifies the type of data and provides variable names for each item included in the list. In this example, the method code updates the employee's last name, but does not return any information to the calling program.

NOTE Java uses the terms *parameters* and *arguments* when working with methods. Some developers use these terms interchangeably. But generally, the term *parameter* is used to refer to the variable listed in the method signature and an *argument* is the value that is passed to the method during program execution.

The parameter list can have zero, one or many parameter values. The values in the list do not have to be the same data type. The parameter list data type must match the data type of the values in the call to the method. In my example, I can only include one String value when I call the method `setEmployeeLastName`.

NOTE If the method does not require any information from the calling program, an empty parameter list must be provided.

4.1.5 Method Body

When a method is called, control is passed to the method where it executes each statement in the method. The method body returns control when one of the following occurs:

- all statements have been executed
- a return statement is encountered
- an error is thrown (which is covered later in the book)

If the method executes successfully, it returns control to the statement immediately following the call to the method. The body of the method has access to the parameters listed in the parameter list and includes a block of code to accomplish some task.

If the method signature has a `void` return type, then the return statement can be omitted in the body of the method. Otherwise, the method body **must** contain a return statement and the value returned must match the return type specified in the method signature.

Quick Check 4-1:

1. T/F: The keyword `void` is used to indicate the method is not used.
2. T/F: A void method must have a return statement.
3. What return type should you use for returning the price of an item?

4. In the following code snippet, match the visibility modifier, return type, method name, and parameter list:

```
public int countSomething(int x, int y, int z) {
```

visibility modifier	countSomething
return type	(int x, int y, int z)
method name	public
parameter list	int

5. Which of the following are valid method names:

- calculate Cost
- moveCharacter
- 1stPlayer

6. What is missing from this method signature:

```
public void printName {
```

4.2 Adding Getter and Setter Methods

The terms getter and setter are used to describe methods that *get* and *set* instance data in a Java class. Remember, instance data is usually declared as *private*, therefore, in order to access this data from within the same class, it is good programming practice to provide a getter and setter method for each piece of instance data.

The benefit of using a setter method is that we can add additional code checking to enforce data integrity. For example, if we have instance data that contains a last name, we might want to add logic to make sure the calling program is not setting it to an empty String.

Let's revisit the Car class from Lesson 3 and update it with a getter and setter method for each piece of instance data. Listing 4.1 shows the code from lesson 3. Remember, each car object was given values for the following instance data:

- Make, Model, Year, Miles Per Gallon (MPG), and Color

Listing 4.1: Car Class

```
1. public class Car {
2.     private String make, model, color;    // #A
3.     private int year;
4.     private double mpg;
5.
6.     public void moveForward() {
7.     }
8.     public void moveBackward() {
```

```

9.     }
10.    public void stop() {
11.    }
12.    public void turnLeft() {
13.    }
14.    public void turnRight() {
15.    }
16.    public void honk() {
17.    }
18.    public static void main(String[] args) {
19.    Car myCar = new Car();
20.        myCar.make = "Subaru";
21.        myCar.model = "Outback";
22.    myCar.year = 2017;
23.    myCar.mpg = 28;
24.    myCar.color = "black";
25.    }
26.    }

```

#A Since the three variables: make, model, and color are all String variables, I put them together separated by a comma

This code works great and demonstrates how to create an object from the Car class. Then it assigns values to each piece of instance data. But it would be easy to make a mistake, such as providing a year < 1900 or an empty String for the car make. Also, since the instance data is marked as private, only code in this class can access these fields. What happens if we want to make this Car class available to other classes in our application?

The solution is to add getter and setter methods. Code listing 4.2 shows the new Car class with a getter and setter method for all the instance data.

Listing 4.2: Car Class with Getter and Setter Methods

```

1.    public class Car {
2.        private String make, model, color;
3.        private int year;
4.        private double mpg;
5.
6.        public String getMake() { return make; }    // #A
7.        public String getModel() { return model; } // #A
8.        public String getColor() { return color; } // #A
9.        public int getYear() { return year; }     // #A
10.       public double getMPG() {return mpg; }      // #A
11.
12.       public void setMake(String make) {         // #B
13.           this.make = make; }
14.       public void setModel(String model) {      // #B
15.           this.model = model; }
16.       public void setColor (String color) {    // #B
17.           this.color = color; }
18.       public void setYear (int year) {
19.           if(year >= 1900 && year <=2050)
20.               this.year = year;
21.           else

```

```

22.         System.out.println("Invalid year");
23.     }
24.     public void setMPG(double mpg) {           // #B
25.         this.mpg = mpg; }                   // #B
26.
27.     public void moveForward() { }
28.     public void moveBackward() { }
29.     public void stop() { }
30.     public void turnLeft() { }
31.     public void turnRight() { }
32.     public void honk() { }
33.     public static void main(String[] args) {
34.         Car myCar = new Car();
35.         myCar.make = "Subaru";
36.         myCar.model = "Outback";
37.         myCar.year = 2017;
38.         myCar.mpg = 28;
39.         myCar.color = "black";
40.     }
41. }

```

#A These are getter methods for each field of instance data

#B The setter methods often include data validation, similar to the check for the car year

Now, the Car class has methods for retrieving the instance data about any car object and updating the data. By making these methods public, they can be used by this program and others. Also, notice that each getter method has a return type, so when the calling program uses these methods, it must expect the value returned to be the same data type.

The setter methods require the calling program to provide the data that will be used to update the instance data, this is provided in the parameter list.

Note:

In each setter method, I had to use the keyword **'this'**. When a method uses a parameter that has the same name as the field for that class, we need to identify which piece of data is part of the object vs. the data included in the parameter list. So, remember, the keyword **'this'** refers to the object that is affected by the calling program. Figure 4.1 shows the use of the **'this'** keyword.

```

class Car {
    private String make, model, color;
    private int year;
    private double mpg;
    public void setMake(String make) {
        this.make = make;
    }
    public void setModel(String model) {
        this.model = model;
    }
    . . .
}

```

Figure 4.2 Shows how to use the keyword **'this'** for updating instance data

Note:

The setter method uses the dot notation which I introduced in Lesson 2. In this example, it is used for accessing the instance data, i.e.: `this.make`. In the next section I will explore more dot notation when I start calling the methods created for the car class.

Now that we have getter and setter methods defined, I will add code to the main method to use these new methods. For this example, I am only going to include the main method in the code listing. The main method will:

Print all the information about myCar

Update all the information about myCar and print the new information

Attempt to update the year of myCar to 2075

Output from executing code in Listing 4.3

```
My car:
Subaru
Outback
2017
28.0
black
```

} CAR OBJECT DATA BEFORE

```
My car:
Ford
Mustang
2018
55.7
Red
Invalid year
```

} CAR OBJECT DATA AFTER

↑

This message is generated when the code tries to set the year to 2075

Figure 4.3 Output listing after executing the code in Listing 4.3

Listing 4.3 Car class using the new getter and setter methods

```
1. public class Car {
2.     private String make, model, color;
3.     private int year;
4.     private double mpg;
5.
6.     //#A
7.     public static void main(String[] args) {
8.         Car myCar = new Car();
9.         myCar.make = "Subaru";
10.        myCar.model = "Outback";
11.        myCar.year = 2017;
12.        myCar.mpg = 28;
```

```

13.     myCar.color = "black";
14.
15.     System.out.println("My car: ");
16.     System.out.println(myCar.getMake()); // #B
17.     System.out.println(myCar.getModel());
18.     System.out.println(myCar.getYear());
19.     System.out.println(myCar.getMPG());
20.     System.out.println(myCar.getColor()); // #B
21.
22.     myCar.setMake("Ford"); // #C
23.     myCar.setModel("Mustang");
24.     myCar.setYear(2018);
25.     myCar.setMPG(55.7);
26.     myCar.setColor("Red"); // #C
27.
28.     System.out.println("My car: ");
29.     System.out.println(myCar.getMake()); // #D
30.     System.out.println(myCar.getModel());
31.     System.out.println(myCar.getYear());
32.     System.out.println(myCar.getMPG());
33.     System.out.println(myCar.getColor()); // #D
34.
35.     myCar.setYear(2075); // #E
36. }
37. }

```

#A Code is omitted to save room, see code listing 4.2 for missing information

#B Each print line statement uses the getter methods to retrieve the instance data for the myCar object

#C Each of these statements update the instance data using the value in the parameter list and calling the setter methods

#D Print the new values for the myCar object after using the setter methods

#E Try to set the year to an invalid year (2075)

The next section discusses how we can modify the private instance data for our objects.

Quick Check 4.2:

1. Methods used to retrieve instance data are called: _____.
2. Which statement is correct for getting the model of myCar:
 - a. `myCar.getModel;`
 - b. `myCar.getModel();`
 - c. `myCar.setModel();`
3. Which statement is correct for getting the mpg for myCar:
 - a. `myCar.getMPG();`
 - b. `myCar.getmpg();`
 - c. `getMPG();`
4. Methods used to **update** instance data are called: _____.
5. Which statement is correct for setting the year of myCar to 1998:

- a. `myCar.setYear(1998);`
- b. `setYear(1998);`
- c. `myCar.getYear(1998);`

6. Which statement is correct for setting the mpg for myCar to 50.5:

- a. `myCar.setmpg(50.5);`
- b. `myCar.setMPG(50.5);`
- c. `myCar.setMPG("50.5");`

4.3 Adding More Methods

In addition to the getter and setter methods, there are other methods that can be added to our program. For example, it might be helpful to have a method that can determine how many miles can be driven with the current fuel in the car tank.

Every car object has information about the MPG rating for the vehicle, so all we really need is to ask the user how much fuel is in the tank. Then we can take the MPG and multiply it times the amount of fuel to determine the distance the car can travel before refueling. For example, if we have a car with an MPG rating of 34 and 10 gallons in the tank, this car can drive 340 miles before it needs to be refueled.

This example needs to know how much fuel is in the tank, so that value must be included in the parameter list for our new method. Since the method is calculating the remaining distance, it will return a numeric value. The instance data field for MPG is a double, so we should expect the final distance to be returned as a double. Here is the method signature and the code needed to calculate the distance:

```
public double calculateDistance(double fuel) {
    double distance = fuel * mpg;
    return distance; }
```

In this code, the method signature has a return type of double, and the parameter list has one value for the amount of fuel in the tank. Inside the method, a new double variable is created called distance that is calculated using the fuel times the mpg. Then, the distance variable is returned. Listing 4.4 shows the new method and new statements in the main method to call this method and print the distance the car can travel.

Listing 4.4: The calculateDistance method is added to the Car class

```
1. import java.util.Scanner;    // #A
2. public class Car {
3.     private String make, model, color;
4.     private int year;
5.     private double mpg;
6.
7.     // getter and setter methods are omitted in this printed version
```

```

8.
9.     public void moveForward() { }
10.    public void moveBackward() { }
11.    public void stop() { }
12.    public void turnLeft() { }
13.    public void turnRight() { }
14.    public void honk() { }
15.
16.    public double calculateDistance(double fuel) {    //#B
17.        double distance = fuel * mpg;              //#C
18.        return distance;                            //#D
19.    }
20.    public static void main(String[] args) {
21.        Scanner in = new Scanner(System.in);        //#E
22.        Car myCar = new Car();
23.        myCar.make = "Subaru";
24.        myCar.model = "Outback";
25.        myCar.year = 2017;
26.        myCar.mpg = 28;
27.        myCar.color = "black";
28.
29.        System.out.println("Enter fuel remaining in tank: ");
30.        double fuelInTank = in.nextDouble();        //#F
31.        double distance = myCar.calculateDistance(fuelInTank);    //#G
32.        System.out.println("The car can travel " + distance + "miles");    //#H
33.
34.    }
35. }

```

#A This import statement provides access to the Scanner class which is used to read and write to the console

#B This method signature indicates that it returns a double and has a double as a parameter

#C Add a new variable to hold the value of mpg * fuel in the tank

#D Return a double value

#E Create a new Scanner object, in, that can be used to read data from the command line

#F Read the value entered by the user at the command line, it is expecting a numeric value that can have a decimal point

#G Call the calculateDistance method on the object myCar, giving it the value for the fuel in the tank and save the value returned in the variable distance

#H Print out the new value for the distance remaining with the current fuel

In this last section, I added a method to calculate how far the car can travel with the current fuel in the tank. This method was added to the Car class, so it is only available to objects created using the Car class. In the main method, I can call (or invoke) this method using the car object. In this example, my car object is called myCar, so I use the dot notation to invoke the method: `myCar.calculateDistance(fuelInTank)`.

Let's look carefully at the call to the method. Did you notice that I have the variable fuel as the argument? The call to the method is not the same as the method signature. The signature provides the instructions for using the method. The call to the method allows me to pass values to the method, but I don't include the data type.

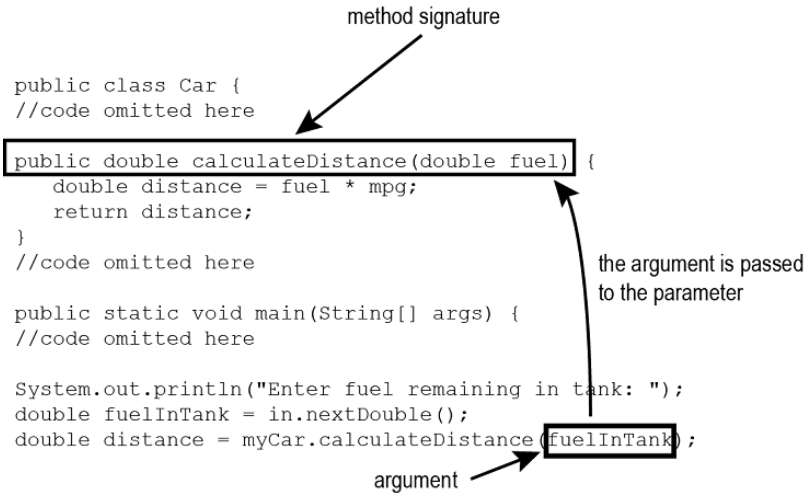


Figure 4.4 Diagram showing how the argument is passed to the method through the parameter list

Quick Check 4-3:

- Given the method signature above, which statement correctly calls the method `calculateMPG` given the fuel used and the distance travel for the object `myCar`:
 - `myCar.calculateMPG();`
 - `myCar.calculateMPG(10, 120);`
 - `calculateMPG(10,120);`
- What value can NOT be returned from calling the method above
 - 12.0
 - "12.0"
 - Twelve

4.4 Summary

In this lesson, you learned:

- how to identify the components of a method signature
- how to add Getter and Setter methods to your class
- how to add more methods to your program

The objective was to add methods to a class. This lesson included special methods called getter and setter methods, and we added one of teach for all the instance data in the `Car` class. Then we added another method to our class that calculates the remaining distance a car can travel based on the fuel remaining in the tank and the car's mpg.

In the next lesson, I will introduce the concept of Arrays and ArrayLists in Java which can be used to create a variable that represents multiple data items.

Try this:

Using the data integrity rules in table 4.3, update the car class to include this logic in the remaining setter.

Table 4.3: Data integrity rules for the car class

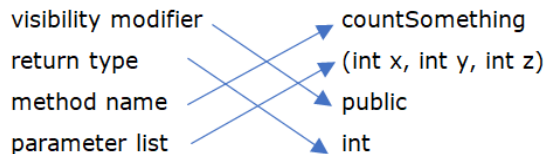
Instance Data	Rule
make	3 <= String length < 50
model	4 < String length < 100
year	1900 < year < 2050
mpg	0 < mpg < 100
color	4 < Strength length < 40

Quick Check 4-1 Solutions:

1. T/F: The keyword `void` is used to indicate the method is not used. **FALSE**
 - a. A void return type indicates that the method does not return any values
2. T/F: A void method must have a return statement. **FALSE**
 - a. The return statement is optional for a method with a return type of void
3. What return type should you use for returning the price of an item?
 - a. `double`, the price should allow for floating point values (values with a decimal point)
4. In the following code snippet, match the visibility modifier, return type, method name, and parameter list:


```
public int countSomething(int x, int y, int z) {
```

```
public int countSomething(int x, int y, int z) {
```



5. Which of the following are valid method names:
 - a. `calculate Cost` (method names cannot have a space)
 - b. **`moveCharacter`**
 - c. `1stPlayer` (method names cannot start with a number)

6. What is missing from this method signature:

```
public void printName {
```

This statement is missing arguments.

Quick Check 4.2, fill in the blank and multiple choice:

1. Methods used to retrieve instance data are called: **getter methods**.
2. Which statement is correct for getting the model of myCar:
 - a. `myCar.getModel;`
 - b. `myCar.getModel();`**
 - c. `myCar.setModel();`
3. Which statement is correct for getting the mpg for myCar:
 - a. `myCar.getMPG();`**
 - b. `myCar.getmpg();`
 - c. `getMPG();`
4. Methods used to **update** instance data are called: **setter methods**.
5. Which statement is correct for setting the year of myCar to 1998:
 - a. `myCar.setYear(1998);`**
 - b. `setYear(1998);`
 - c. `myCar.getYear(1998);`
6. Which statement is correct for setting the mpg for myCar to 50.5:
 - a. `myCar.setmpg(50.5);`
 - b. `myCar.setMPG(50.5);`**
 - c. `myCar.setMPG("50.5");`

Quick Check 4-3 solution:

```
public double calculateMPG(double fuelUsed, double distanceTravelled)
```

1. Given the method signature above, which statement correctly calls the method `calculateMPG` given the fuel used and the distance travel for the object `myCar`:
 - a. `myCar.calculateMPG();`
 - b. `myCar.calculateMPG(10, 120);`**
 - c. `calculateMPG(10,120);`
2. What value can be returned from calling the method above:
 - a. 12.0**
 - b. "12.0"

c. twelve

Solution to Try This Exercise:

Listing 4.5 Add data validation logic to the setter methods for the Car class

```

1.  public class Car {
2.      private String make, model, color;
3.      private int year;
4.      private double mpg;
5.
6.      public String getMake() { return make; }
7.      public String getModel() { return model; }
8.      public String getColor() { return color; }
9.      public int getYear() { return year; }
10.     public double getMPG() {return mpg; }
11.
12.     public void setMake(String make) {
13.         if(make.length() >= 3 && make.length() < 50)  //A
14.             this.make = make;
15.         else
16.             System.out.println("Invalid make");}
17.     public void setModel(String model) {
18.         if(model.length() > 4 && model.length() < 100)  //B
19.             this.model = model;
20.         else
21.             System.out.println("Invalid model");}
22.     public void setColor (String color) {
23.         if(color.length() > 4 && color.length() < 40)  //C
24.             this.color = color;
25.         else
26.             System.out.println("Invalid color");}
27.     public void setYear (int year) {
28.         if(year >= 1900 && year <=2050)  //D
29.             this.year = year;
30.         else
31.             System.out.println("Invalid year");
32.     }
33.     public void setMPG(double mpg) {
34.         if(mpg >= 0 && mpg < 100)  //E
35.             this.mpg = mpg;
36.         else
37.             System.out.println("Invalid mpg");}
38.
39.     // remaining code omitted

```

#A Validation check added to check the length of the String is between 3 and 49 characters.

#B Validation check added to check the length of the String is between 4 and 99 characters.

#C Validation check added to check the length of the String is between 4 and 39 characters.

#D Validation check to make sure year is valid (between 1900 and 2050)

#E Validation check added to make sure the numeric value for mpg is between 0 and 99.

5

Adding Loops

After reading lesson 5, you will be able to:

- Understand the syntax of loops in Java
- Choose the appropriate type of loop for your program
- Use the for-each loop (or enhanced for loop) to iterate over a series of values

A loop allows us to repeat a block of code until some condition is met. In Java, the types of loops include:

for loop – used when you know how many times the loop needs to execute

while loop – most common and flexible loop, continues while a condition is true

do...while loop – same as a while loop, but the test condition is executed at the end of the loop, therefore the code is always executed at least once

for-each – iterates over a series of values without any counter or sentinel value

This lesson reviews each type of loop and examples of each loop. Each type of loop has some common errors that can cause an endless loop. This is referred to as an infinite loop and sometimes the cause is hard to find. So, for each loop I will review some common errors that can cause an infinite loop.

Consider This

Have you ever gone shopping at the grocery store and only had a few items so you decide to check out using the lane marked 'Less than 15 items' only to realize the person in front of you definitely has more than 15? What would happen if the grocery store had a program that only allowed up to 15 items when using this checkout lane? What if the person in line counted incorrectly and they actually have 16 items?

Well, lucky for those people who don't always follow the rules, most stores do not have these limits set in the system. ~~but~~ What kind of loop can we use when we don't know how many items the customer is purchasing?

5.1 Adding a Count Controlled Loop

Have you ever used the pre-programmed buttons on your microwave oven? One of my favorite buttons on the microwave is the popcorn button. The programming for this button starts a timer that counts down the time required to heat up a bag of popcorn. Counters can either count up to a value or count down to a value. In this case, the counter is used to count down the time to the value zero when it stops the heating process.

When learning Java after learning another language such as Python, the **for** statement might seem a little difficult to understand, so let's take a look at each part of the **for** loop. Figure 5.3 takes the **for** loop statement needed for our program and labels each section of the statement.

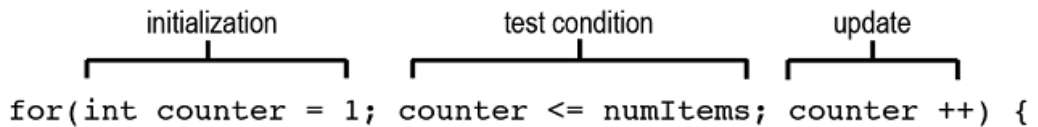


Figure 5.1: Diagram of the syntax of a for loop in Java

Every **for** loop has three sections:

Initialization – This section is only executed once at the beginning of the loop execution. It provides the value of the control variable, in our case we are using counter and setting it to the value one. The statement ends with a semi-colon. (Note: More than one variable can be defined in this section.)

Test Condition – This part of the for loop is used to control the number of iterations using a boolean expression. As long as the test condition evaluates to true, the loop will continue. Once the expression is false, the loop ends. This part of the loop is evaluated each time the loop executes. This statement also ends at the semicolon.

Update – To make sure the loop eventually ends, this section is usually used to update the counter variable so that the test condition evaluates to false when the number of iterations have been completed.

This diagram shows the flow of logic in a for loop in Java. It starts with the initialization, then the test condition. If the condition is true, then it executes the block of code. When the condition changes to false, the loop is finished, and the program continues with the next statement outside of the for loop.

The update statement is used to change the control variable. This value can be changed by incrementing it by one, decrementing it by one, and even increment/decrement by some other value, such as by twos.

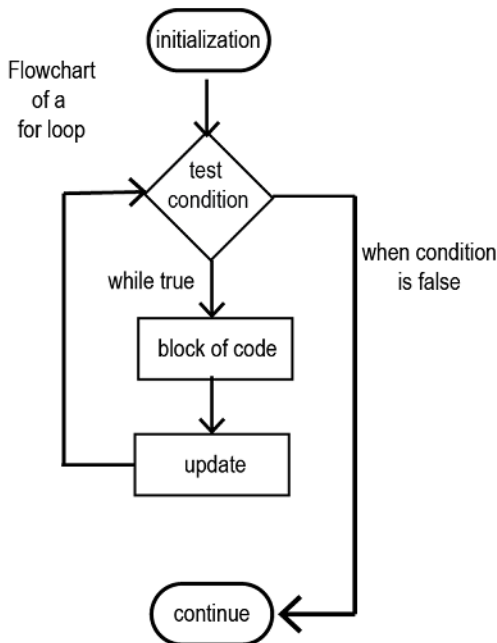


Figure 5.2: A flowchart depicting the process used in a for loop

Now that we have reviewed the syntax for a count-controlled loop, let's look at a code snippet that replicates the counter for microwaving a bag of popcorn. Note: In this example, the counter is actually starting at 180 and counting down to zero.

Listing 5.1 This is a code snippet for counting down from 180 seconds to zero

```

for(int time = 180; time > 0; time--)
{
    System.out.print(time + ", ");
}
System.out.println("The popcorn is done!! Let's eat!");
  
```

When using a for loop, be careful to make sure that you are updating your control variable correctly. In this example, if we accidentally increased the time in the for statement, the number would keep getting bigger and the loop would never end. Here is an example:

```

for(int time = 180; time > 0; time++)
  
```

By using `time++`, each time the loop executes it would add one to the time variable and it would never be less than or equal to zero so it would never stop.

NOTE: Another type of error that can occur when using a for loop is to accidentally add a semicolon after the for statement. The semicolon indicates the end of a statement, so the program would execute the for statement, then continue to the code inside the for loop but only execute it one time. Here is an example:

```
for(int time = 180; time > 0; time --);
{ System.out.print("counting down"); }
```

This code would only print the message "counting down" once.

Quick Check 5-1:

1. Write a code snippet using a for loop that prints all the numbers from 1 to 100 (inclusive)
2. Reverse the loop to print the numbers starting at 100 down to 1
3. Challenge: Write a code snippet to print every other number from 1 to 100 (hint, use the update portion of the for loop to increase the number by two instead of one)
4. What output is printed for each of these statements:

```
a) for(int i = 0; i<5; i++){System.out.println(i);}
b) for(int j = 5; j>=0; j--) {System.out.println(i);}
```

5.2 The While Loop

Section 5.1 describes the process for adding a count-controlled loop which is great when you know how many times to execute your block of code, but that is not always the case. The while loop is probably the most commonly used loop. The idea is that the block of code is executed while the condition is true. This implies that something in the block of code must change the condition to false in order to end the loop.

The generic flow of a while loop starts at the top of the code, tests if the condition is true. If the condition is true, it executes the block of code and returns to the top where it tests the condition again. Figure 5.4 is a flowchart of a generic while loop.

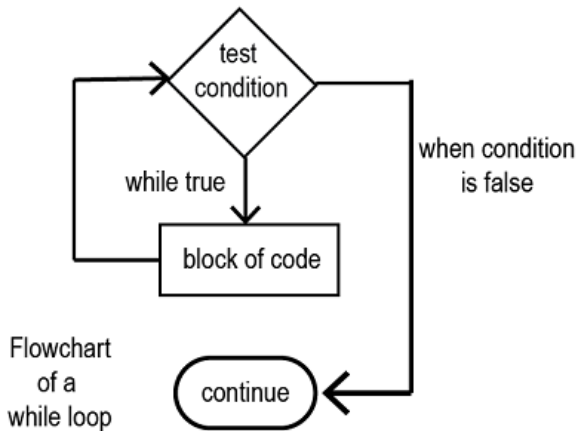


Figure 5.3: A flowchart depicting the flow of a while loop in Java

If the condition never changes to false, this causes an infinite loop. Another common error when using a while loop is to accidentally add a semicolon after the while statement, this always causes an infinite loop.

Every for loop can be rewritten as a while loop. Consider the popcorn example from section 5.1. We can start by declaring a timer variable and set it to 180. Then, the test condition for the while loop can check if the value is greater than zero. Inside the while loop, the timer is decreased by one each time. Here is a code snippet for this while loop:

Listing 5.2: Code snippet to countdown from 180 to zero using a while loop

```
int time = 180;
while(time > 0)
{
    System.out.print(time + ", ");
    time = time - 1;
}
System.out.println("The popcorn is done!! Let's eat!");
```

For this code snippet, there are at least two ways to easily cause an infinite loop. The first is to accidentally add a semicolon after the while test condition, so it looks like this:

```
while(time > 0);
```

Remember, in Java the semicolon indicates the end of a statement. So, in this case the time variable starts at 180 and is never changed so it is always greater than zero.

The second situation that causes an infinite loop is forgetting to change the counter inside the loop body. Listing 5.3 shows the code snippet for an infinite loop.

Listing 5.3: Code snippet with an error that causes an infinite loop

```
int time = 180;
while(time > 0)
{
    System.out.print(time + ", ");
}
System.out.println("The popcorn is done!! Let's eat!");
```

The variable to keep track of the `time` starts at 180, but it is never changed inside the body of the while loop. Therefore, this loop repeats until the user cancels the program or it runs out of memory.

Quick Check 5-2:

1. What is wrong with this code statement:

```
while(done == true); {...}
```

2. Change each **for** loop to a **while** loop:

a. `for(int i = 0; i<10; i++) {...}`

b. `for(int j = 10; j>=0; j--) {...}`

5.3 The Do...While Loop

Have you ever written a program that allows the user to choose a value from a list, such as a menu? This situation represents a business requirement where the best loop is probably the do...while loop. The do...while loop always executes the code inside the loop at least once and then the test condition is at the end. Figure 5.5 shows the flowchart for a do while loop.

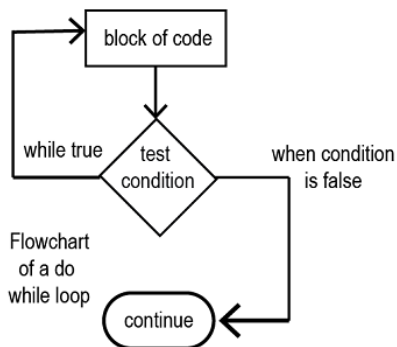


Figure 5.4: A flowchart for a do...while loop

The code for this type of loop is very similar to a while loop, except the test condition is at the end so the loop always executes at least once. Here is an example of a program that prints a menu and asks the user to choose from a list of options.

Listing 5.4: Use a Do...While Loop to prompt the user with a menu

```

1. package fruit;
2. import java.util.Scanner;
3. public class Fruit {
4.
5.     public static void main(String[] args) {
6.         Scanner in = new Scanner(System.in);
7.         System.out.println("What is your favorite summer fruit?");
8.         int choice = 0;
9.         do {
10.            System.out.println("1. Apples");
11.            System.out.println("2. Oranges");
12.            System.out.println("3. Pears");
13.            System.out.println("4. Blueberries");
14.            System.out.println("Choose one: ");
15.            choice = in.nextInt();
16.        } while(choice > 4);
17.        System.out.println("You chose: " + choice);
18.    }
19. }

```

In this example, the program presents the user with a list of options. Then, when the user enters a number, it checks to make sure it is not greater than four since there are only four options. If the number is greater than four, the condition evaluates to true and the menu is presented to the user again until they select a number from 1 to 4.

Quick Check 5-3:

1. Which statement is true for a do...while loop:
 - a. the loop needs to execute at least once
 - b. you use it when you know how many times you need to execute the loop
 - c. you use it when the program needs to iterate over every element in a collection
2. T/F: A do...while loop checks the test condition first.

5.4 The for-each Loop

The last type of loop that is used in Java is called an enhanced for loop, or a for-each loop. This loop automatically iterates over a collection of items. To demonstrate the for-each loop, I must introduce a collection of items. In Java, one of the simplest ways to create a collection of elements is using an array. An array is used to hold a collection of items that are all the same type. In Java, the array syntax uses square brackets, which creates a variable called `videoGames` that holds a collection of String elements.

To loop through all the items, we can use a for loop, here is an example:

Listing 5.5: Using a For Loop to Print a List of Video Games

```
String[] videoGames = {"Super Mario Odyssey", "Call of Duty: WWII", "Splatoon 2"};
for(int i = 0; i<3; i++) {
    System.out.println(v[i]);
}
```

Now, the enhanced for loop uses a simpler and easier to read syntax. The enhanced for loop is a great choice when you need to access every element in a collection of elements. The bad part of this type of loop is that you cannot identify where an element exists in a list. For example, if you wanted to only print the third element you have to use a traditional for loop. Listing 5.5 creates an array that contains three string values with the names of three video games. Then the for-each loop prints out each string. Notice the syntax for the for-each loop, it starts with the keyword `for`, then in parentheses it has the data type of the elements in the list followed by a variable name, `v`. Next is a colon and the name of the list, `videoGames`. The code inside the curly brackets has access to the variable `v`. In this case, I just printed each value. Figure 5.7 depicts the parts of a for-each loop. The data type must match the type of values in the collection.

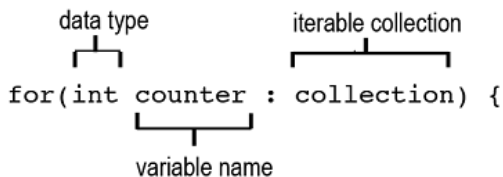


Figure 5.5: Diagram of the components in a for-each loop

Listing 5.6: Using an Enhanced For Loop to Print a List of Video Games

```
String[] videoGames = {"Super Mario Odyssey", "Call of Duty: WWII", "Splatoon 2"};
for(String v:videoGames) {
    System.out.println(v);
}
```

In each of the loop examples in this lesson I used curly brackets to enclose the code that is executed inside the loop. If there is only one statement that needs to be executed in the loop, then the curly brackets are optional. So, the following two code snippets will produce the same results.

```
for(int i = 0; i<10; i++) {           // #A
    System.out.println(i);
}                                     // #B

for(int i = 0; i<10; i++)
    System.out.println(i);           // #C
```

- #A This line has an open curly bracket to indicate the start of the code to be executed inside this loop
- #B This is the closing curly bracket for the block of code
- #C Since there is only one statement, the curly brackets are optional

This is only true when you have one statement that needs to be executed inside the loop!

Quick Check 5-4:

1. What will the following code snippet print?

```
String[] names = {"Mario", "Princess Peach", "Luigi", "Bowser"};
for(String n:names) {
    System.out.println(n);
}
```

2. Given this list of values:

```
String colors = {"Red", "Orange", "Yellow", "Green", "Blue", "Indigo",
                "Violet"};
```

write an enhanced for loop to print the colors of the rainbow.

5.5 Summary

In this lesson, we learned:

the syntax for each type of loop in Java

when to use each type of loop

how to use an enhanced for loop to iterate over a series of elements

There are many times when you need to repeat a block of code. This is usually the time when you will choose a loop to control how many times to repeat a block of code. If you know the number of times you need to repeat these statements, then you can use a for loop. A while loop is used when it is not clear how many times you want to execute the loop, such as entering prices for items at a grocery store, you don't know how many items the customer has ahead of time. The do...while loop is used when you want to execute the block of code at least once. And finally, use the for-each loop when the program needs to access every element in a collection.

In the next lesson, I introduce a capstone project that provides an opportunity to explore the concepts taught in this unit.

Try This:

Use NetBeans to create a new project. In the main method of the .java file, add a loop to print the numbers starting from 10 to 1, print each number on a separate line. When the loop finishes print "blastoff!". Run the program to check your code.

Quick Check 5-1:

1. Write a code snippet using a for loop that prints all the numbers from 1 to 100 (inclusive)

```
for(int i = 1; i <= 100; i++) {System.out.println(i);}
```

2. Reverse the loop to print the numbers starting at 100 down to 1

```
for(int i = 100; i > 0; i--) {System.out.println(i);}
```

3. Challenge: Write a code snippet to print every other number from 1 to 100 (hint, use the update portion of the for loop to increase the number by two instead of one)

```
for(int i = 1; i <= 100; i += 2) {System.out.println(i);}
```

4. What output is printed for each of these statements:

a) `for(int i = 0; i<5; i++){System.out.println(i);}`

0

1

2

3

4

b) `for(int j = 5; j>=0; j--) {System.out.println(i);}`

5

4

3

2

1

0

Quick Check 5-2:

1. What is wrong with this code statement:

```
while(done == true); {...}
```

This statement causes an infinite loop. The problem is that there is a semi-colon immediately following the for loop condition.

2. Change each for loop to a while loop:

a. `for(int i = 0; i<10; i++) {...}`

Solution:

```
int i = 0;
while(i < 10) {...
i++; }
```


b. `for(int j = 10; j>=0; j--) {...}`

Solution:

```
int j = 10;
while(j >= 0) {...
j--; }
```

Quick Check 5-3:

1. Which statement is true for a do...while loop:
 - a. **the loop needs to execute at least once** (correct)
 - b. you use it when you know how many times you need to execute the loop
 - c. you use it when the program needs to iterate over every element in a collection
2. T/F: A do...while loop checks the test condition first. (**false**)

Quick Check 5-4:

1. What will the following code snippet print?

```
String[] names = {"Mario", "Princess Peach", "Luigi", "Bowser"};
for(String n:names) {
    System.out.println(n);
}
```

Output:

```
Mario
Princess Peach
Luigi
Bowser
```

2. Given this list of values:

```
String colors = {"Red", "Orange", "Yellow", "Green", "Blue", "Indigo",
"Violet"};
```

write an enhanced for loop to print the colors of the rainbow.

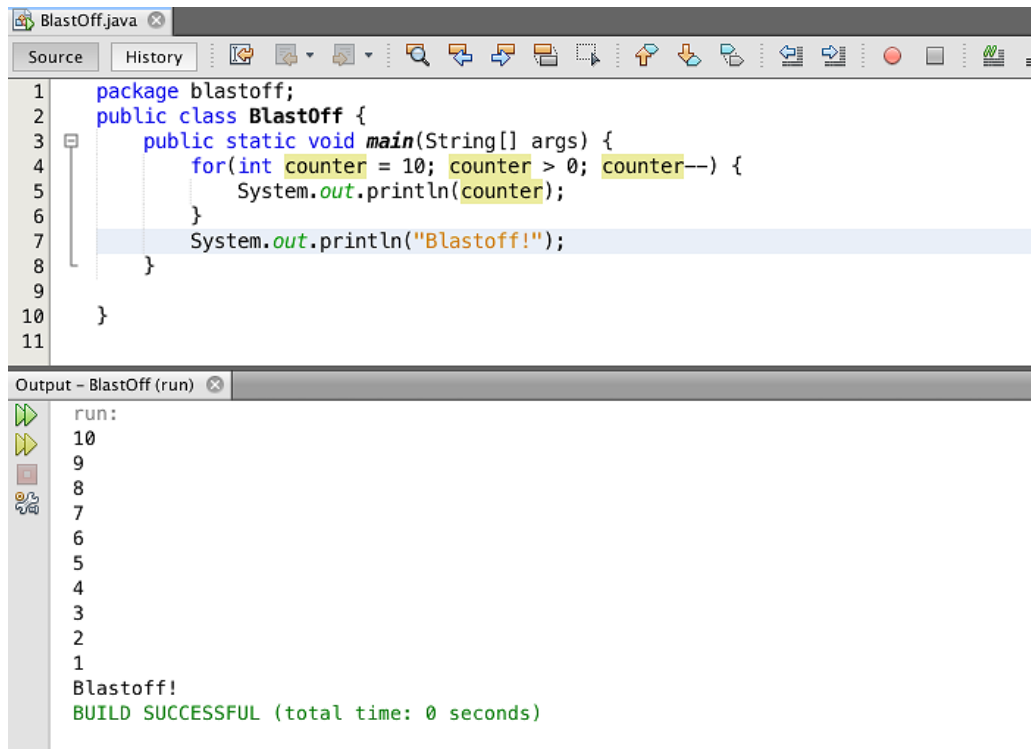
```
for(String c:colors) {
    System.out.println(c);
}
```

Try This Solution:

Use NetBeans to create a new project. In the main method of the .java file, add a loop to print the numbers starting from 10 to 1, print each number on a separate line. When the loop finishes print "blastoff!". Run the program to check your code.

Solution:

This activity can be completed using either a for loop or a while loop. Since I want to print the numbers from 10 to 1, I can use a for loop and print the counter value each time. Figure 5.8 shows a sample solution to this problem.



```
1 package blastoff;
2 public class BlastOff {
3     public static void main(String[] args) {
4         for(int counter = 10; counter > 0; counter--) {
5             System.out.println(counter);
6         }
7         System.out.println("Blastoff!");
8     }
9
10 }
11
```

Output - BlastOff (run)

```
run:
10
9
8
7
6
5
4
3
2
1
Blastoff!
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 5.6: Screenshot of a program to print numbers from 10 to one, then Blastoff!

6

Arrays and ArrayLists

This lesson covers:

- Creating arrays to hold a list of data elements
- Accessing and modifying elements in an array
- Manipulating arrays using the Arrays class
- Creating ArrayLists to hold a list of objects
- Accessing and modifying elements in an ArrayList
- Understanding the difference between an array and ArrayList

An array is an object that is used to hold a fixed list of data elements (each item in an array is referred to as an element) that are the same data type. They are extremely useful and often used in OO programming. You can think of an array as a list of data elements. The Java collections API has a list interface that is discussed in a later chapter of this book.

An array can be a list of numbers, Strings, characters, or even a list of boolean values. In Java, Strings act like primitive data types but they are actually objects. So, an array can store a list of primitive data or a list of objects. For example, an array can be used to hold the responses of a student who took a true/false quiz, which would be a list of boolean values. If the quiz had 25 questions, without an array, we would need 25 variables to hold the student responses. By using an array, we can hold all 25 values with a single variable.

Consider This

Have you ever purchased a loaf of sliced bread at the grocery store? It usually comes in a single package with individual slices packed inside. What would happen if we needed to have each slice packaged separately? The individual slices are similar to creating separate variables for a list of data elements that are all the same data type. Instead, it would be better to use a single package, in programming terms, an array, to keep track of each individual element with only one variable.

Can you think of other items that would be better if we could package them together instead of individually?

6.1 Creating arrays

In Java, an array of elements can be stored with a single variable name. When creating an array, it must be defined with a data type and a specific size for the array. In our quiz example, the array is declared as `boolean` with 25 elements, here is the syntax for creating that array:

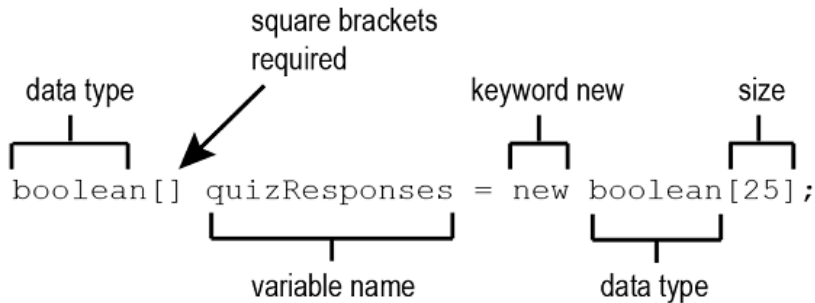


Figure 6.1 This diagram describes the syntax of using an array to hold 25 boolean values.

It is possible to declare the array in one statement and then assign the size in a second statement:

```
boolean[] quizResponses;
quizResponses = new boolean[25];
```

Note:

Arrays are declared with a **fixed size**. An array cannot automatically expand or shrink depending on the number of elements added or removed from the array. Using a fixed size is helpful if you know the specific number of elements, it can save processing time and is very efficient for data access. But if the array needs to grow, it requires the creation of a new larger array and then copying the current list to the new array so additional elements can be added. This is very inefficient and significantly increases processing time.

Table 6.1 shows examples of each data type:

Table 6.1 Examples of arrays with each of the primitive data types in Java

Data type	Array declaration
<code>int</code>	<code>int[] numStudents = new int[20];</code>
<code>double</code>	<code>double[] itemCost = new double[30];</code>

char	char[] alphabet = new char[26];
String	String[] names = new String[25];
boolean	boolean[] responses = new boolean[25];
float	float[] averages = new float[15];
long	long[] bigNumbers = new long[10];
short	short[] smallNumbers = new short[15];
byte	byte[] rgbValues = new byte[6];

When an array is created, it is considered an object that contains the address of the first element in the array. Each subsequent element is stored sequentially in memory. Let's take a look at a diagram of an array declared as an `int` array with 5 elements:

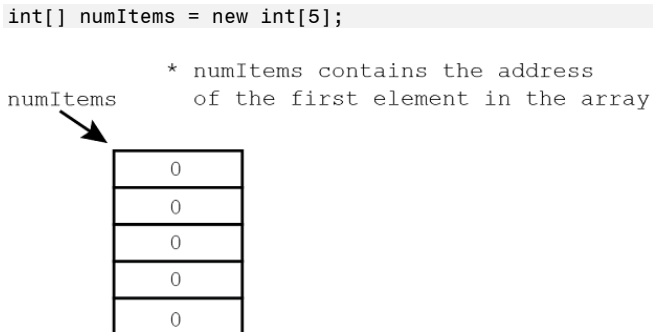


Figure 6.2 This diagram shows how the array `numItems` is stored in memory.

Notice that each element contains the value zero since the array was initialized as an `int` data type. In Java, integer variables are automatically assigned the value zero unless the initial declaration assigns a specific value. Table 5.2 shows the default values for each primitive data type.

Table 6.2 Default values for array elements

Data type	Array values
<code>int</code> , <code>short</code> , <code>byte</code> , <code>long</code>	<code>0</code>
<code>float</code> , <code>double</code>	<code>0.0</code>
<code>char</code>	null character <code>'\u0000'</code>
<code>String</code>	null
<code>boolean</code>	<code>false</code>

Quick Check 6-1:

1. T/F: Once an array is declared, the size cannot be changed.
2. Which statement correctly creates an array of size 100 for a list of integers:
 - a. `int[100] values = new int[];`
 - b. `int[] values = new int[100];`
 - c. `int values = new int[100];`
3. What information is stored in the variable name of an array:
 - a. first element in the array
 - b. address (or reference) value of the array
 - c. variable name of the array

6.2 Access and modify elements in an array

Once the array is created and it is populated with data elements, we need to access the elements in the array. Each element in the array is numbered sequentially, but there is a catch. Array elements start at the element in position 0 and end with the last element which is found using the array length - 1. Let's revisit our `numItems` array, but this time, I've added the location of each element, this value is referred to as the **index** of the element in the array.

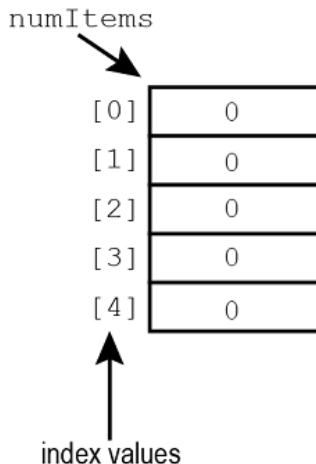


Figure 6.3 In this diagram, the index values of each element have been added.

As you can see in this diagram, the first element is stored in position 0 and the last element is stored in position 4. Notice that the length of this array is 5, so the last element is at the index value of: $5 - 1 = 4$.

When creating an array, the length of the array is automatically stored and accessible through the `length` property. For example, in the diagram 5.3, we can obtain the size using the `numItems.length` statement. It is important to note that the length is the allocated length and does not reflect how many elements have values or not.

When creating an array, it can be declared as an empty array that populates the elements later in the code, or we can provide initial values for our array. So far, I've shown several examples of creating arrays without providing values for the elements in the array. Now, here are some examples of declaring an array with a starting list of elements:

Table 6.3 This table shows arrays that are initialized with values and the length of each array.

Data type	Array declaration	length
int	<code>int[] numStudents = {10, 12, 10, 9};</code>	4
double	<code>double[] itemCost = {3.25, 5.50, 7.75};</code>	3
char	<code>char[] alphabet = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'};</code>	26
String	<code>String[] names = {"Kendrick", "Kathleen"};</code>	2
boolean	<code>boolean[] responses = {true, false, false, true, true};</code>	5
float	<code>float[] averages = {23.5, 45.9, 82.90};</code>	3
long	<code>long[] bigNumbers = {987654321, 876543219};</code>	2
short	<code>short[] smallNumbers = {14, 28, 34};</code>	3
byte	<code>byte[] rgbValues = {127, -127};</code>	2

Note:

Remember, all elements in the array must be the same data type and to find the size of the array, use the dot notation to access the length property of the array (for example: `numItems.length`).

Now that we have created some arrays, let's look closer at how to access specific elements in an array and how to modify elements in an array. It is important to remember that the index of the elements in an array starts at zero and ends at `length - 1`.

Using the example for an integer array in table 5.2, here is how that data is stored:

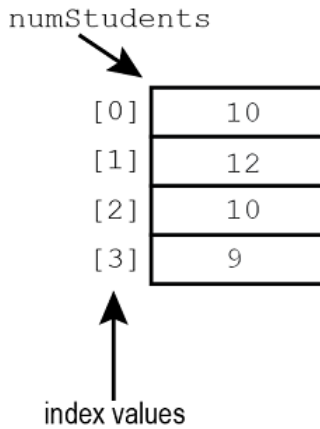


Figure 6.4 This diagram shows the structure of an integer array called `numStudents` with the index values for each item in the array.

This array represents the number of students in four separate sections of an English class. If we have a new student who just started, we need to increase the number of students for the new student's section. If the student is in the third section, we need to change the value from 10 to 11, in other words, add one to the value at index 2 (remember, the indices start at zero, so the third value is in index value 2).

```
numStudents[2] = numStudents[2] + 1;
```

To access an element in an array, we start with the array name, and then place the index of the element in square brackets. For this example, I am changing the value in `numStudents` at index 2 to be increased by 1.

Note:

A common mistake when working with arrays is to forget that the first index starts at zero. Then, if you try to access the last element, your index value will be one more than the last valid index value and this causes an out of bounds exception error. For example, if the array `numStudents` has 4 elements, this statement causes an error:

```
numStudents[4] = numStudents[4] - 1;
```

Here is the error message:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4 at
ArrayExample.main(ArrayExample.java:104)
```

Note that the error message provides the index value that caused the out of bounds error. In this case, it was the value 4, but the last index value is 3.

To traverse every element in an array, it is a common programming practice to use a for loop. The for loop provides an easy way to evaluate every element in an array. Here is an example of code that prints every element of an array:

Listing 6.1 Example of a for loop that prints all letters in the array alphabet

```

1. char[] alphabet = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
   'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'};
2.
3. for(int i = 0; i < alphabet.length; i++)    // #A
4.     { System.out.print(alphabet[i]); }      // #B
5.     System.out.println();

```

#A This for loop traverse each element in the alphabet array, starting at zero and stopping at length-1

#B The print statement uses the loop control variable, i, to print each element

Without arrays, this code would require 26 variables, one for each letter of the alphabet. But with arrays, we can use one variable name and store all values in the array with one variable name.

The for loop starts by printing the element at position zero and stops at one less than the length of the array. A for loop can also be used to populate an array. For example, here is code that creates an integer array and populates the array with even numbers from 0 to 100 (not including 100).

Listing 6.2 This code creates an integer array containing even numbers from 0 – 100

```

1. int index = 0;                                // #A
2. int[] evenNumbers = new int[50];             // #B
3. for(int i = 0; i < 100; i++)                 // #C
4.     { if(i % 2 == 0) {                       // #D
5.         evenNumbers[index] = i;             // #E
6.         index++;                             // #F
7.     }
8. }

```

#A It is important to have a value to track the location of the element in the array

#B Create an array to hold 50 integers

#C Use a for loop to evaluate each number between 0 and 100

#D Check if dividing by 2 produces no remainder and therefore the value is even

#E If the value is even, then store that value in the next open spot in the array

#F Increase the index by one to the next open spot in the array

NOTE: In Java, the symbol % is used for modulus division. Modulus division divides two integers and only returns the remainder portion of the operation. For example, 4 % 2 yields zero, but 5 % 2 yields 1.

The next section shows how to create an ArrayList to hold a list of objects.

Quick Check 6.2:

Given the array: `double[] prices = {10.50, 5.75, 13.90, 15, 14.85};`

1. What is the value of `prices.length`:
 - a. 4
 - b. 6
 - c. 5

2. What is the index value of the element containing the value 15:
 - a. 3
 - b. 4
 - c. 2

3. What is printed using this statement: `System.out.println(prices[5]);`
 - a. 14.85
 - b. Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
 - c. 0

4. Which statement correctly changes the price of item 2 (5.75) to 15.75:
 - a. `prices[2] = 15.75;`
 - b. `prices[1] = 15.75;`

array elements cannot be changed

6.3 Manipulating arrays using the Arrays class

With the popularity of arrays, it is no surprise that there is an `Arrays` class as part of the standard Java API in the `util` package and can be accessed with this import statement:

```
import java.util.Arrays;
```

The `Arrays` class provides numerous methods that make manipulating arrays easy. This particular class is a static class, which means that it is not invoked using dot notation on an object, instead, it can be used to perform operations on one or more arrays. Some examples of the types of manipulations available through this class include:

- `binarySearch`
- `copyOf`
- `copyOfRange`
- `equals`
- `fill`
- `sort`
- `and toString`

Each of these operations has multiple variations, but they make working with arrays much easier. For example, to print the contents of an array, I can use the `toString` method. Figure 6.5 shows the output of printing an array with and without using the `Arrays` class.

Figure 6.5 Screenshot of output showing two ways to print an array variable

The code listing used to demonstrate the use of the `Arrays` class is in Listing 6.3.

Listing 6.3 This code prints an array with and without using the `Arrays` class

```

1. package test;
2. import java.util.Arrays; //A
3. public class ArrayAPIExample {
4.     public static void main(String[] args) {
5.         double[] prices = {10.50, 5.75, 13.90, 15, 14.85};
6.         System.out.println("Print the value of the prices array "
7.             + "without using the Arrays class: " + prices); //B
8.         System.out.println("Print the value of the prices array
9.             + "using the Arrays class: " + Arrays.toString(prices)); //C
10.    }

```

#A This line uses the `import` statement to provide access to the `Arrays` class

#B Since an array variable contains the reference address of the first element in the array, the value printed with this statement is just a hexadecimal representation of a memory address

#C Using the `Arrays` class and the `toString` method, it is easy to print all elements in the array

For more information about the `Arrays` class, go to: <https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html>

Quick Check 6-3:

Here is an excerpt from the site above for the `Arrays` class:

```

static void    sort(int[] a)
                Sorts the specified array into ascending numerical order.

```

1. Given an `int` array, `ages`, which statement correctly sorts it in ascending order:
 - a. `sort(ages);`
 - b. `Arrays.sort(ages);`
 - c. `ages.Arrays.sort;`

6.4 Creating ArrayLists

In Java, arrays are commonly used to hold primitive data types and each array is a fixed size. When a program needs to store a list of objects, it is more common to use an `ArrayList`. The `ArrayList` holds a collection of objects and it is resizable!

An ArrayList can dynamically change size, which is a great feature, but it comes at a cost. When an array is declared, it must specify a size, but because arrays are a fixed size based on the array type declaration, they do not require as much memory as their ArrayList counterparts.

In this section, I will provide a brief overview of ArrayLists so we can use them to hold objects, but later in the book, Lesson 26 Collections where I will provide even more detail about ArrayLists and their uses in OO programming.

Similar to arrays, when creating an ArrayList we start with the data type. The big difference is that an ArrayList can only be used with class data types. In lessons 2-4, I introduced a Car class and each time I wanted to create a car object, I had to create a new variable. Now, I can use an ArrayList to hold a list of Car objects with only one variable. Here is an example:

```

class name      keyword      indicates that this is
  |             |             an empty ArrayList
ArrayList<Car> usedCars = new ArrayList<Car> ();
  |             |             |
  |             |             |
  |             |             |
variable name  class name

```

Figure 6.5 This diagram identifies the parts of an ArrayList used to create a new empty ArrayList of car objects

The ArrayList in figure 6.6 contains a list of Car objects. The syntax used to denote the type of data in the ArrayList is called a diamond operator and was introduced with Java 7. When using the diamond operator, the compiler only requires the diamond on the left to include a value. So, we could rewrite this statement as:

```
ArrayList<Car> usedCars = new ArrayList<>();
```

This structure is used with Generics which is addressed later in this book.

The variable, `usedCars`, is essentially a reference to the start of a list of Car objects.

In the next section, I will add elements, modify existing elements, remove elements and access elements in an ArrayList.

Quick Check 6-4:

1. Which statement creates an ArrayList using the Person class:

- a. `ArrayList<Person> employees = new Person();`
- b. `ArrayList<Person> employees = ArrayList<Person>();`
- c. `ArrayList<Person> employees = new ArrayList<Person>();`

6.5 Accessing and modifying elements in an ArrayList

I stated earlier that ArrayList is a class. It is included in the Java.util library, which means that you need to add an import statement to the top of your code:

```
import java.util.ArrayList;
```

This class comes with the Java API and it includes methods that can be used to add elements, remove elements, get elements, and even set elements in the list. There are quite a few methods available, but let's start with a list of the most commonly used methods with examples of each method.

Table 6.5 Commonly used methods for an ArrayList

Returns	Method	Description
boolean	add(element)	Adds the element to the end of the list
element	get(index)	Returns the element found at the index specified
element	set(index, element)	Replaces the element at the index specified, returns the element that was replaced
int	size()	Returns the number of elements in the list
element	remove(index)	Removes the element at the index position

Now that I have reviewed how to create an ArrayList and provided some of the methods for an ArrayList, let's modify the main method in our Car class to add an ArrayList of car objects. Table 6.6 shows the information for each car. For this activity, I will create an ArrayList of used cars, add car objects to the list, print the number of cars in the list and then remove a car from the list.

Table 6.6 shows the information for three used car objects

object name	make	model	year	mpg	color
oldCar1	Subaru	Outback	2017	28	black
oldCar2	Chevrolet	Malibu	2015	30	grey
oldCar3	Ford	Fusion	2013	32	blue

These are all the cars in the usedCars ArrayList

```

These are the cars on the lot:
[
  Car make: Subaru      Car model: Outback      Car year: 2017  MPG: 28.0      Color: black
,
  Car make: Chevrolet   Car model: Malibu       Car year: 2015  MPG: 30.0      Color: grey
,
  Car make: Ford        Car model: Fusion       Car year: 2013  MPG: 32.0      Color: blue
]
There are 3 used cars on the lot

```

A car is removed using usedCars.remove(0);

These are all the cars in the usedCars ArrayList after removing the car at index zero

```

These are the cars on the lot after removing the first element in the list:
[
  Car make: Chevrolet   Car model: Malibu       Car year: 2015  MPG: 30.0      Color: grey
,
  Car make: Ford        Car model: Fusion       Car year: 2013  MPG: 32.0      Color: blue
]
There are 2 used cars on the lot

```

To print the number of cars, I used the statement usedCars.size()

Figure 6.6 This is the output from the code in listing 6.4 which uses an ArrayList to hold a list of used cars.

Listing 6.4 Car class that uses an ArrayList to hold a list of used cars

```

public class Car {
...  //#A
}
public static void main(String[] args) {
  Scanner in = new Scanner(System.in);
  ...  //#B

  ArrayList<Car> usedCars = new ArrayList<Car>();  //#C
  usedCars.add(oldCar1);  //#D
  usedCars.add(oldCar2);  //#D
  usedCars.add(oldCar3);  //#D

  System.out.println("These are the cars on the lot: ");
  System.out.println(usedCars);  //#E
  System.out.println("There are " + usedCars.size() +  //#F
    " used cars on the lot");
  usedCars.remove(0);  //#G
  System.out.println("\nThese are the cars on the " +
    "lot after removing the first element in the list: ");
  System.out.println(usedCars);  //#E
  System.out.println("There are " + usedCars.size() +  //#F
    " used cars on the lot");
}

```

#A Car class is omitted, see Lesson 4 for details of Car class
 #B Code to create car objects is omitted
 #C Create the usedCar ArrayList
 #D Add the car object to the end of the list
 #E Print all the cars in the list
 #F Use the .size() method to get the number of cars in the list
 #G Remove the first car from the list

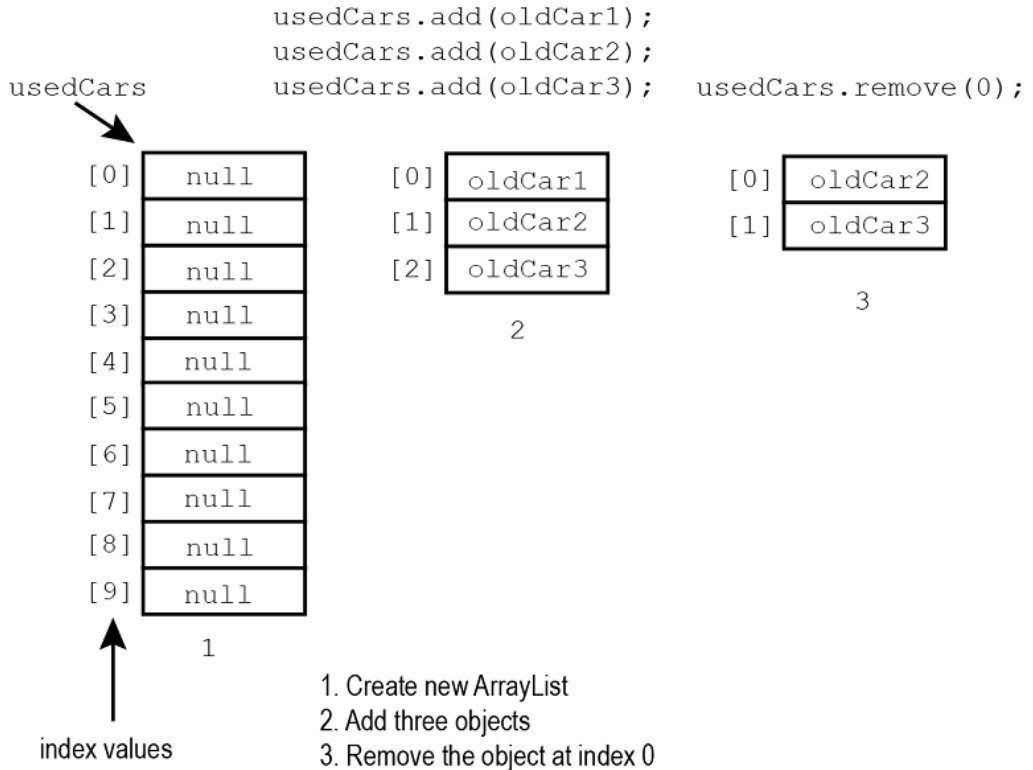


Figure 6.7 Shows the stages of the usedCar ArrayList as cars are added and removed

Quick Check 6-5:

For these questions, please review to the coding listing 6.4 and the output in figure 6.7

1. What make and model of car is returned from the statement: `usedCars.remove(0)`;
 - a. Chevrolet Camaro
 - b. Toyota Camry
 - c. Dodge Dart
2. For the next few questions, consider the following code is added to the end:

```
Car myCar = new Car();
myCar.setMake("Lexus");
myCar.setModel("CT");
myCar.setColor("Stratus Gray");
myCar.setMPG(43);
myCar.setYear(2013);
```

2. If I added the statement: `usedCars.add(myCar)`, where would this car get added to the list?
 - a. Beginning
 - b. End
 - c. Causes an Index Out of Bounds Exception

3. What is the size of the `ArrayList` after I add `myCar` to the list:
 - a. 4
 - b. 3
 - c. 5

6.6 Summary

In this lesson, you learned:

- The syntax for creating arrays to hold a list of data elements
- How to access and modify elements in an array
- The syntax for creating `ArrayLists` to hold a list of objects
- How to access and modify elements in an `ArrayList`
- The difference between an array and `ArrayList`

In this lesson you learned how to create and use both arrays and `ArrayLists` to hold lists of data. Remember, arrays are a fixed size and can only be used with primitive data. `ArrayLists` can only hold objects but they are much more flexible for adding/removing items from the list.

In the next lesson, I will introduce a problem and a solution that uses all the topics from Lessons 1-6.

Try this:

Using the concepts from the previous lessons, create a `Student` class that includes the student first and last name, email, and course. Next, create an `ArrayList` of all students in the course. Print out the list of students and the number of students in the course.

Try using some of the `ArrayList` methods to add students, remove students at various locations in the `ArrayList`, and even change the student at the beginning of the list using the `set` method.

Quick Check 6-1 Solution:

1. T/F: Once an array is declared, the size cannot be changed.
2. Which statement correctly creates an array of size 100 for a list of integers:
 - a. `int[100] values = new int[];`
 - b. `int[] values = new int[100];`
 - c. `int values = new int[100];`

3. What information is stored in the variable name of an array:
 - a. first element in the array
 - b. **address (or reference) value of the array**
 - c. variable name of the array

Quick Check 6.2 Solution:

Given the array: `double[] prices = {10.50, 5.75, 13.90, 15, 14.85};`

1. What is the value of `prices.length`:
 - a. 4
 - b. 6
 - c. **5**
2. What is the index value of the element containing the value 15:
 - a. **3**
 - b. 4
 - c. 2
3. What is printed using this statement: `System.out.println(prices[5]);`
 - a. 14.85
 - b. **Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException**
 - c. 0
4. Which statement correctly changes the price of item 2 (5.75) to 15.75:
 - a. `prices[2] = 15.75;`
 - b. `prices[1] = 15.75;`
 - c. array elements cannot be changed

Quick Check 6-3 Solution:

Here is an excerpt from the site above for the Arrays class:

```
static void      sort(int[] a)
                Sorts the specified array into ascending numerical order.
```

1. Given an int array, `ages`, which statement correctly sorts it in ascending order:
 - a. `sort(ages);`
 - b. **`Arrays.sort(ages);`**
 - c. `ages.Arrays.sort;`

Quick Check 6-4 Solution:**1. Which statement creates an ArrayList using the Person class:**

- a. `ArrayList<Person> employees = new Person();`
- b. `ArrayList<Person> employees = ArrayList<Person>();`
- c. `ArrayList<Person> employees = new ArrayList<Person>();`

Quick Check 6-5 Solution:

For these questions, please review to the coding listing 6.4 and the output in figure 6.7

1. What make and model of car is returned from the statement: `usedCars.remove(0)`:

- a. Chevrolet Camaro
- b. Toyota Camry
- c. **Dodge Dart**

2. For the next few questions, consider the following code is added to the end:

```
Car myCar = new Car();
myCar.setMake("Lexus");
myCar.setModel("CT");
myCar.setColor("Stratus Gray");
myCar.setMPG(43);
myCar.setYear(2013);
```

3. If I added the statement: `usedCars.add(myCar)`, where would this car get added to the list?

- a. Beginning
- b. **End**
- c. Causes an Index Out of Bounds Exception

5. What is the size of the ArrayList after I add `myCar` to the list:

- a. **4**
- b. 3
- c. 5

Solution to Try This Exercise:**Listing 6.5 Car class with an ArrayList for newCars**

```
1. import java.util.ArrayList;
2. import java.util.Scanner;
3. public class Student {
4.     private String fName, lName, email, courseName;
5.     public String getfName() {
6.         return fName; }
7.     public void setfName(String fName) {
```

```

8.         this.fName = fName; }
9.     public String getlName() {
10.         return lName; }
11.     public void setlName(String lName) {
12.         this.lName = lName; }
13.     public String getEmail() {
14.         return email; }
15.     public void setEmail(String email) {
16.         this.email = email; }
17.     public String getCourseName() {
18.         return courseName; }
19.     public void setCourseName(String courseName) {
20.         this.courseName = courseName; }
21.     public String toString() {
22.         return fName + " " + lName + ", email: " + email + ", course: " +
23.             courseName ;
24.     }
25.     public static void main(String[] args) {
26.         Scanner in = new Scanner(System.in);
27.         String firstName, lastName, emailAddress, course;
28.         boolean done = false;
29.
30.         ArrayList<Student> roster = new ArrayList<Student>(); // #A
31.         while (!done) {
32.             System.out.println("Enter first name, last name, email, and " +
33.                 "course name ");
34.             firstName = in.nextLine();
35.             lastName = in.nextLine();
36.             emailAddress = in.nextLine();
37.             course = in.nextLine();
38.             Student s = new Student(); // #B
39.             s.setfName(firstName); // #B
40.             s.setlName(lastName); // #B
41.             s.setEmail(emailAddress) // #B
42.             s.setCourseName(course); // #B
43.             roster.add(s); // #C
44.             System.out.println("Want to enter another student? (y/n) ");
45.             if (in.nextLine().equalsIgnoreCase("n")) {
46.                 done = true;
47.             }
48.         }
49.         System.out.println("These are the students in the course: ");
50.         System.out.println(roster);
51.         System.out.println("There are " + roster.size() + " students\n");
52.         roster.remove(0); // #D
53.         Student newStudent = new Student();
54.         newStudent.setfName("Pat");
55.         newStudent.setlName("Slattery");
56.         newStudent.setEmail("patSlattery@aol.com");
57.         newStudent.setCourseName("English");
58.         roster.set(1, newStudent); // #E
59.         System.out.println("Students remaining: ");
60.         System.out.println(roster);
61.         System.out.println("There are " + roster.size() + " students\n");
62.     }
63. }

```

- #A Create an ArrayList to hold the list of student objects
- #B Create a new student and populate the instance data
- #C Add the new student to the end of the ArrayList
- #D Remove the first student in the list
- #E Change the student at index 1 (the second occurrence) to a new student

7

Capstone 1

In lessons 1 – 6, I introduced the NetBeans Integrated Development Environment (IDE), reviewed the concept of creating classes to represent real world objects, how to add data for each object, how to add methods for each object, and how to modify the data associated with a specific object.

This lesson is a capstone of these topics providing you with the opportunity to reinforce the skills learned. In this lesson, I will introduce a problem that is designed to include everything presented so far in the book.

In this lesson, I will walk through the process of creating a simple payroll application. Let's start by thinking about the problem, what are we trying to accomplish? The program needs to create the weekly payroll for each employee. Sounds simple, but what about overtime? How about employees who are paid annually? First, it is important to gather the business requirements for our problem.

For this problem, I will act as the company owner. Here are the requirements for this assignment:

- The payroll report prints the employee first name, last name, and weekly earnings
- Hourly employees earn overtime for any hours greater than 40 in one week
- Overtime hours are paid at a rate of 1.5 times their regular hourly rate
- Salaried employees are paid their annual salary divided by 52 each week

NOTE: Some years have 53 weeks in a year, but this program will keep it simple and use **only 52 weeks**.

Now that I have explained our problem, let's walk through a solution using Java. The next section describes the process of flowcharting our business problem to be used as a guide for writing code for our final solution.

7.1 Flowchart the Solution

Creating a flowchart of the process before you start coding is a great way to think through your solution to a problem. Start by putting yourself in the position of a payroll processor. Here is a flowchart of the problem:

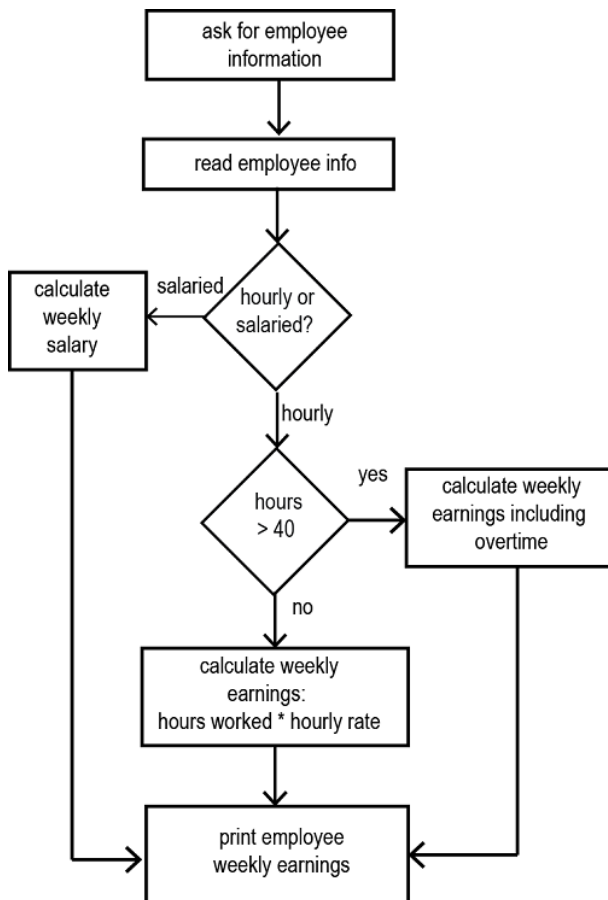


Figure 7.1 Flowchart of the payroll processing program

For this example, the payroll application is pretty simple. If the employee is paid hourly, the application asks for the hourly wage and hours worked. Otherwise, the annual salary is entered for the employee. Based on the employee type, the weekly payroll is calculated. In the next few sections, I will review the Java code necessary for creating a complete payroll program.

7.2 Create an Employee Class

The Employee class contains information about the employee and methods that describe the behavior of an employee object. Each employee object has the following data fields:

- First name
- Last name
- Type (hourly or salaried)
- Hourly rate
- Hours worked
- Salary

Listing 5.1 shows the code necessary for the class and the instance data.

Listing 7.1 The Employee class with instance data

```

1. class Employee {
2.     private String firstName, lastName;    // #A
3.     private char employeeType;
4.     private double hourlyWage, salary, hoursWorked = 0;
5.     ...
6. }
```

#A Multiple variables can be declared in one statement, separated by commas

The Employee class now has the necessary instance data, the next section adds the getter and setter methods for each instance data field.

7.3 Add getter and setter methods to the Employee class

Since all the instance data for this class is declared as private, the only way to update these fields is by using a series of methods that can retrieve the data (getter) and modify the data (setter). This is important to make sure that fields are not accidentally updated with invalid data, such as an employee type that is not 'h' or 's' since they are the only two valid values..

So, for each piece of instance data, I am adding the getter and setter method. Here is the revised code including these methods. Note: these methods are declared as public so the main program can access them.

Listing 7.2 Adding getter and setter methods to the Employee class

```

1. class Employee {
2.     private String firstName, lastName;    // #A
3.     private char employeeType;           // #A
4.     private double hourlyWage, salary, hoursWorked = 0; // #A
5.     public void setFirstName(String fName) { // #B
6.         firstName = fName; } // #B
7.     public String getFirstName() { // #B
8.         return firstName; } // #B
9.     public void setLastName(String lName) { // #B
```

```

10.     lastName = lName; }                               //#B
11.     public String getLastName() {                     //#B
12.         return lastName; }                           //#B
13.     public void setEmployeeType(char type) {          //#B
14.         if(type != 'h' && type != 's')
15.             System.out.println("Invalid Employee Type, must be s or h");
16.         else                                          //#B
17.             employeeType = type; }                    //#B
18.     public char getEmployeeType() {                   //#B
19.         return employeeType; }                       //#B
20.     public void setHourlyWage(double hWage) {         //#B
21.         hourlyWage = hWage; }                       //#B
22.     public double getHourlyWage() {                   //#B
23.         return hourlyWage; }                         //#B
24.     public void setSalary(double salary) {           //#B
25.         this.salary = salary; }                      //#B
26.     public double getSalary () {                     //#B
27.         return salary; }
28.     public void setHoursWorked (double hoursWorked) {
29.         this.hoursWorked = hoursWorked; }           //B
30.     public double getHoursWorked () {
31.         return hoursWorked; }
32. }

```

#A Start by identifying the instance data for the object

#B For each instance data, create a getter and setter method to retrieve and change the instance data. If the variable name in the parameter list is exactly the same as the instance data variable name, use the this operator to identify the instance data field rather than the parameter data field

7.4 Add methods to the Employee class

In this section I will review the process for adding additional methods to our Employee class that can be used to calculate the weekly earnings and print the weekly payroll information for each employee. The process for calculating weekly earnings is different if the employee is hourly or salaried. If the employee is hourly, there is an additional check to see if they worked more than 40 hours. If the employee worked more than 40 hours, then they get paid overtime for any hours greater than 40. The hourly wage for the overtime hours is 1.5 times the regular hourly wage. Listing 7.3 shows the code with the new method that calculates the weekly payroll for each employee.

Listing 7.3 Add a method to calculate the weekly payroll for an employee

```

1.     //code omitted
2.     protected double calculateEarnings() {
3.         double weeklyEarnings;
4.         if (employeeType == 'h') {                   //#A
5.             if (hoursWorked <= 40) {
6.                 weeklyEarnings = hoursWorked * hourlyWage;
7.             } else {
8.                 weeklyEarnings = 40 * hourlyWage + ((hoursWorked - 40)
9.                 * hourlyWage * 1.5);
10.            }

```



```

11.     } else {                               //#B
12.         weeklyEarnings = salary / 52;
13.     }
14.     return weeklyEarnings;
15. }

```

#A This code checks if the employee is paid hourly (if the type = 'h')

#B This else statement indicates that this employee is not hourly, therefore they must be salaried

The next method that I want to add is a way to format the output for printing the employee information including the weekly payroll. Every class automatically has a toString() method, but it does not format the information the way we want. So, in Listing 7.4 I have added a new toString() method for our Employee class.

Listing 7.4 Add a method to print the employee information

```

1.     class Employee {
2.         private String firstName, lastName;
3.         private char employeeType;
4.         private double hourlyWage, salary, hoursWorked = 0;
5.
6.         // Getter and Setter methods are omitted, see code above
7.
8.         public double calculateWeeklyEarnings() {
9.             if(employeeType == 'h') {
10.                 if(hoursWorked <= 40)
11.                     return hoursWorked * hourlyWage;
12.                 else
13.                     return 40 * hourlyWage +
14.                        ((hoursWorked - 40) * (1.5 * hourlyWage));
15.             }
16.             //this is a salaried employee
17.             return salary/52;
18.         }
19.         public String toString() {
20.             return "First Name: " + getFirstName() +
21.                "\t\tLast Name: " + getLastName() +                //#A
22.                "\nHourly Rate: " + getHourlyWage() +                //#B
23.                "\tTotal wages: $" + calculateWeeklyEarnings() +
24.                "\n";
25.             "Total wages: $" + getTotalWages();
26.         }
27.     }

```

#A The \t is an escape sequence and causes the output to include a tab character sequence

#B The \n is also an escape sequence that is used to print the string on the next line

7.5 Wrap It Up

Now that the Employee class is created, we can go back to the main method and start adding employees. To allow for multiple employees, I am using an ArrayList of type Employee to hold the list of employees. The program starts by asking the user to enter the employee

information and then indicate when they are done by answering the query: Do you have more employees to enter? (y/n).

After all the employees have been stored in the ArrayList, I will iterate through the list and print out the weekly earnings for each employee. Listing 7.5 shows the complete application.

Listing 7.5 Complete payroll application

```

1.  import java.util.ArrayList;
2.  import java.util.Scanner;
3.
4.  class Employee {
5.      private String firstName, lastName;
6.      private char employeeType;
7.      private double hourlyWage, salary, hoursWorked = 0;
8.      public void setFirstName(String fName) {
9.          firstName = fName; }
10.     public String getFirstName() {
11.         return firstName; }
12.     public void setLastName(String lName) {
13.         lastName = lName; }
14.     public String getLastName() {
15.         return lastName; }
16.     public void setEmployeeType(char type) {           // #A
17.         if (type != 'h' && type != 's') {
18.             System.out.println("Invalid Employee Type, must be s or h");
19.         } else {
20.             employeeType = type;
21.         }
22.     }
23.     public char getEmployeeType() {
24.         return employeeType; }
25.     public void setHourlyWage(double hWage) {
26.         hourlyWage = hWage; }
27.     public double getHourlyWage() {
28.         return hourlyWage; }
29.     public void setSalary(double salary) {
30.         this.salary = salary; }
31.     public double getSalary() {
32.         return salary; }
33.     public void setHoursWorked(double hoursWorked) {
34.         this.hoursWorked = hoursWorked; }
35.     public double getHoursWorked() {
36.         return hoursWorked; }
37.     public double calculateWeeklyEarnings() {         // #B
38.         if (employeeType == 'h') {
39.             if (hoursWorked <= 40) {
40.                 return hoursWorked * hourlyWage;
41.             } else {
42.                 return 40 * hourlyWage + ((hoursWorked - 40) * (1.5 * hourlyWage));
43.             }
44.         } else {
45.             return salary / 52;
46.         }
47.     }
48.     public String toString() {                       // #C

```

```

49.
50.         return "First Name: " + getFirstName()
51.             + "\tLast Name: " + getLastName()
52.             + "\nHourly Rate: " + getHourlyWage()
53.             + "\tTotal wages: $" + calculateWeeklyEarnings()
54.             + "\n";
55.     }
56.
57.     public static void main(String[] args) {
58.         Scanner in = new Scanner(System.in);
59.         String fName, lName, response = "y", hourly = "";
60.         char type;
61.         double hrlyWage = 0, hrsWorked = 0, salary = 0;
62.         ArrayList<Employee> employees = new ArrayList<Employee>();//#D
63.         while (response.equalsIgnoreCase("y")) {
64.             System.out.println("Enter first name: ");
65.             fName = in.nextLine();
66.             System.out.println("Enter last name: ");
67.             lName = in.nextLine();
68.             System.out.println("Is this employee paid hourly? (y/n)");
69.             hourly = in.nextLine();
70.             if (hourly.equalsIgnoreCase("y")) {
71.                 System.out.println("Enter hourly wage: ");
72.                 hrlyWage = in.nextDouble();
73.                 System.out.println("Enter hours worked: ");
74.                 hrsWorked = in.nextDouble();
75.                 type = 'h';
76.             } else {
77.                 System.out.println("Enter yearly salary: ");
78.                 salary = in.nextDouble();
79.                 type = 's';
80.             }
81.             Employee tempEmployee = new Employee();
82.             tempEmployee.setEmployeeType(type);
83.             tempEmployee.setFirstName(fName);
84.             tempEmployee.setLastName(lName);
85.             if (type == 'h') {
86.                 tempEmployee.setHourlyWage(hrlyWage);
87.                 tempEmployee.setHoursWorked(hrsWorked);
88.             } else {
89.                 tempEmployee.setSalary(salary);
90.             }
91.             in.nextLine();
92.             employees.add(tempEmployee);
93.             System.out.println("Are there more employees? (y/n)");
94.             response = in.nextLine();
95.         }
96.         for (Employee e : employees) { // #E
97.             System.out.println(e.toString());
98.         }
99.     }
100. }

```

#A Several of the setter methods have logic to validate the data before making the change

#B This method is used to calculate the employee weekly salary

#C Adding a toString method helps print the object in a readable format

#D Create an ArrayList of type Employee to keep a list of employee objects

#E Use an enhanced for loop to print each employee

It is always important to test your program. So, here is a scenario for a local pizza owner, Paolo, who is the owner of Paolo's Original Italian Pizza. He has two full time employees that are salaried, and five employees that are paid hourly.

Using table 7.1, I will execute our code and print each employee's weekly earnings.

Table 7.1 Contains sample employee information for Paolo's OIP

First name	Last name	Type	Salary	Hourly Wage	Hours worked	Weekly Paycheck
Jose	Garcia	s	56,680			1,090.00
Anna	Marino	s	63,960			1,230.00
Frank	Gallo	h		12.00	37	444.00
Gianna	Hopple	h		12.50	30	375.00
Marco	Ricci	h		13.00	45	617.50
Maria	Giovanni	h		12.75	40	510.00
Roberto	Lombardi	h		14.00	50	770.00

```
First Name: Jose                Last Name: Garcia
Hourly Rate: 0.0              Total wages: $1090.0
```

```
First Name: Anna                Last Name: Marino
Hourly Rate: 0.0              Total wages: $1230.0
```

```
First Name: Frank                Last Name: Gallo
Hourly Rate: 12.0             Total wages: $444.0
```

```
First Name: Gianna                Last Name: Hopple
Hourly Rate: 12.5            Total wages: $375.0
```

```
First Name: Marco                Last Name: Ricci
Hourly Rate: 13.0            Total wages: $617.5
```

```
First Name: Maria                Last Name: Giovanni
Hourly Rate: 12.75           Total wages: $510.0
```

```
First Name: Roberto                Last Name: Lombardi
Hourly Rate: 14.0            Total wages: $770.0
```

Figure 7.2 Screenshot of the output from our payroll program

7.6 Summary

In this lesson, I reviewed the initial business problem of creating a payroll application, used a flowchart to diagram our potential solution, and then walked through the code necessary for each part of the flowchart.

The requirements for this project included:

- The payroll report prints the employee first name, last name, and weekly earnings
- Hourly employees earn overtime for any hours greater than 40 in one week
- Overtime hours are paid at a rate of 1.5 times their regular hourly rate
- Salaried employees are paid their annual salary divided by 52 each week

TRY THIS:

A friend has started a new business selling produce grown in her own garden. Some items are sold by the pound, while others are priced based on quantity. For example, tomatoes are 1.99 per pound, but zucchinis are 2 for 1.00.

Create a `Produce` class that helps your friend easily calculate the total due for each customer. Some customers are purchasing multiple items, so use an `ArrayList` to keep track of all the produce they purchase during one transaction, then print out the total due.

Add the capability to enter the amount tendered by the customer and calculate the change as a little challenge to this activity.

Try this solution:

Listing 7.6 Complete produce class

```

1.  import java.util.ArrayList;           // #A
2.  import java.util.Scanner;
3.
4.  public class Produce {
5.      private String productName;
6.      private double cost;
7.      public String getProductName() {
8.          return productName; }
9.      public void setProductName(String productName) {
10.         this.productName = productName; }
11.     public void setItemCost(double itemCost) {
12.         this.cost = itemCost; }
13.     public String toString() {
14.         return productName + ", " + cost;
15.     }
16.
17.     public static void main(String[] args) {
18.         Scanner in = new Scanner(System.in);
19.         String product;

```

```

20.     int howMany;
21.     double costPerPd, weight, cost, price, total=0,
22.           amtTendered=0, change=0;
23.     boolean done = false;
24.     String response;
25.
26.     ArrayList<Produce> produce = new ArrayList<Produce>();    // #B
27.
28.     while (!done) {
29.         System.out.println("Produce name: ");
30.         product = in.nextLine();
31.         System.out.println("Is this priced per pound? (y/n)");
32.         response = in.next();
33.         if(response.equalsIgnoreCase("y"))
34.             {
35.                 System.out.println("Enter price: ");
36.                 costPerPd = in.nextDouble();
37.                 System.out.println("Enter weight: ");
38.                 weight = in.nextDouble();
39.                 cost = weight * costPerPd;
40.                 howMany = 0;
41.             }
42.         else
43.             {
44.                 System.out.println("Enter price per item: ");
45.                 price = in.nextDouble();
46.                 System.out.println("Enter quantity: ");
47.                 howMany = in.nextInt();
48.                 cost = howMany * price;
49.                 costPerPd = 0;
50.                 weight = 0;
51.             }
52.     }
53.
54.     Produce p = new Produce();    // #C
55.     p.setItemCost(cost);          // #D
56.     p.setProductName(product);    // #D
57.     produce.add(p);              // #E
58.     in.nextLine();
59.     System.out.println("Want to enter another item? (y/n) ");
60.
61.     if (in.nextLine().equalsIgnoreCase("n")) {
62.         done = true;
63.     }
64. }
65. System.out.println(produce);    // #F
66. for(Produce p: produce) {    // #G
67.     total += p.cost;
68. }
69. System.out.println("You owe: $" + total);
70. System.out.println("Enter amount tendered: ");
71. amtTendered = in.nextDouble();
72. change = amtTendered - total;
73. System.out.println("Change: " + change);
74.
75. }
76. }

```

#A An import is required to use the ArrayList class from the Java API
#B Create an empty ArrayList to hold the Produce objects
#C Create a new instance of the Produce class
#D Update the product name and price for the current produce item
#E Add the object just created to the end of the ArrayList
#F Print the produce information, this statement uses the toString method defined in the Produce class
#G Use an enhanced for loop to add the cost of all produce items in this transaction

Unit 2

Application Programming Interface (API)

8

Standard Java API

After reading lesson 8, you will be able to:

- Read the Java Standard API documentation
- Navigate the documentation to find various classes
- Understand when to use the Scanner class from the Java API
- Understand when to use Wrapper classes from the Java API

The Java API, Application Programming Interface, contains the set of classes and interfaces included with the Java Development Environment that are used to build application software. Each class is written in Java and runs on the JVM, Java Virtual Machine.

The idea of the API is to take commonly used classes and make them available to all developers, so they don't need to recreate the code required for these common tasks.

This lesson introduces the topic of the Standard Java API and provides examples of some of the more commonly used classes included in the library.

Consider This

Have you recently purchased a new smart phone? Or even a new laptop or desktop? Each of these products come with pre-installed software and apps. So, instead of writing an app to keep track of your contact list, you can use the contact app that was already pre-installed. There are probably numerous tools on your device that were available when you first started using the device.

Cell phone and computer manufacturers have decided that some tasks are commonly used by all owners of the products. So instead of having each owner program their own OS or program the features that allow you dial numbers and make a call, they come preinstalled.

The Standard Java API includes many classes that a developer normally needs for a software application. For example, the Scanner class allows us to read data entered from the console window. Another example is the Math class, this class contains methods for finding the square root of a number or the result of raising a number to a power. Can you

think of other common tasks that might be required by most developers? Check out the Oracle documentation to see if that feature exists.

8.1 How to read the API documentation

In the Java API, many of the classes are grouped together into packages. We have seen some of these packages in our applications such as the use of an ArrayList or every time we use the Scanner class. Both of these classes are included in the java.util package.

Here is a list of some packages that are commonly used in Java programming:

java.applet – provides the classes needed to create an applet

java.awt – contains classes for creating graphical user interfaces and painting graphics

java.io – used for input and output through data streams, serialization and the file system

java.lang – provides classes necessary to the design of the Java programming language

java.util – contains the collections framework, date and time, and other utility classes

Oracle maintains an extensive set of documentation for the Java API. This is a great resource and extremely useful for using the various classes and interfaces included in the API. The most recent version (during the writing of this text) can be found at this url: <https://docs.oracle.com/javase/8/docs/api/>

The screenshot shows the Java Platform Standard Edition 8 API Specification page. The page is titled "Java™ Platform, Standard Edition 8 API Specification". It includes a navigation bar with "OVERVIEW", "PACKAGE", "CLASS", "USE", "TREE", "DEPRECATED", "INDEX", and "HELP". Below the navigation bar, there is a "PREV" and "NEXT" button, and "FRAMES" and "NO FRAMES" options. The main content area is titled "Java™ Platform, Standard Edition 8 API Specification" and includes a description: "This document is the API specification for the Java™ Platform, Standard Edition." Below this, there is a "See: Description" link. The "Profiles" section lists three profiles: compact1, compact2, and compact3. The "Packages" section is a table with two columns: "Package" and "Description".

Package	Description
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User

Figure 8.1 A screenshot of the Java Platform Standard Edition 8 API Specification

To better understand how to use this documentation, let's drill down on one of the packages. Since we have already used the `ArrayList` class in our last unit to hold a list of objects, we will start with this class.

There are several options for finding the specified class. If you are not sure which package contains the class, use the list of All Classes in the left column, shown in Figure 8.1. For the `ArrayList`, I know that it is included in the `java.util` package. So, on the top left I chose `java.util`, then in the list of classes, I chose `ArrayList` and the main panel now shows the `ArrayList` class.

The screenshot shows the Java Platform Standard Ed. 8 IDE interface. On the left, the package hierarchy is visible, with `java.util` selected. Below it, a list of classes is shown, with `ArrayList` selected. The main panel displays the documentation for the `ArrayList` class. The class signature is `Class ArrayList<E>`. The inheritance hierarchy is shown as `java.lang.Object`, `java.util.AbstractCollection<E>`, `java.util.AbstractList<E>`, and `java.util.ArrayList<E>`. The class implements the `Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `List<E>`, and `RandomAccess` interfaces. The direct known subclasses are `AttributeList`, `RoleList`, and `RoleUnresolvedList`. The class definition is shown as `public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable`. A detailed description explains that it is a resizable-array implementation of the `List` interface, implementing all optional list operations and permitting all elements, including `null`. It notes that the class is roughly equivalent to `Vector`, except that it is unsynchronized. Performance characteristics are also mentioned, stating that `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time, while the `add` operation runs in amortized constant time, and adding `n` elements requires $O(n)$ time.

Figure 8.2 A screenshot of the `ArrayList` class documentation inside the `java.util` package

The next step is to scroll down in the main panel to see the list of methods that are included for this class. Figure 8.3 is a screenshot of the main panel where I scrolled down to the section on Constructors and Method Summary. The constructors are also methods and these can be used to create an `ArrayList` object or instance. The first constructor creates an empty list with an initial capacity of ten. The second constructor creates an `ArrayList` that starts with an initial collection of elements. The last constructor provides the ability to create an `ArrayList` with an initial capacity other than the default of ten.

The method summary section lists all methods available when using an `ArrayList` in our application. The code for implementing these actions has already been written and tested so

it is now available to all developers. This eliminates the individual developer from reinventing the wheel (or the code in this case) for adding an element to the `ArrayList`.

The first four methods in the list are variations of adding elements to the list. The first item includes a capital E and a small e to represent the `Element` class being added. Remember, only objects can be added to an `ArrayList`. In our `Car` class from lesson 5, to add an object to the `ArrayList`, we used:

```
usedCars.add(oldCar1);
```

This is an example of using the method `add(E e)` from this list. Notice that there is a return type listed for this method, `boolean`. It returns the value `true` if the element was successfully added.

Generic class

If you look at figure 8.2, you will see the notation: Class `ArrayList<E>`, where the value E inside of the diamond (< >) acts as a placeholder for the specific class type. This notation is used to indicate a generic class. Generics enable classes to be parameters when defining other classes. For example: `ArrayList<String>` names, uses the `ArrayList` class to create a list of objects that are instances of the `String` class. Unit 4 has a detailed lesson on Generics in Java.

Non-static methods require an object to act upon. But static classes are independent of any specific instance of the class. The `Math` class is a great example of static methods. The `Math` class has numerous static methods, for example, if we want to find the square root of 160, we don't have to create an object to use the `Math` class. Instead, we can use dot notation on the class itself:

```
Math.sqrt(160);
```

Constructor Summary

Constructors

Constructor and Description
ArrayList() Constructs an empty list with an initial capacity of ten.
ArrayList(Collection<? extends E> c) Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.
ArrayList(int initialCapacity) Constructs an empty list with the specified initial capacity.

Method Summary

All Methods	Instance Methods	Concrete Methods
Modifier and Type	Method and Description	
boolean	add(E e) Appends the specified element to the end of this list.	
void	add(int index, E element) Inserts the specified element at the specified position in this list.	
boolean	addAll(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.	
boolean	addAll(int index, Collection<? extends E> c)	

Figure 8.3 A screenshot of the documentation for the methods included in the ArrayList class

Listing 8.1 shows an example of code that uses an ArrayList to hold a list of ice cream flavors. In this code snippet, I have created an initial list of flavors and then created a second ArrayList with two more flavors, add this second list to the original list. A String object represents a sequence of characters. In the java.lang package there are several interesting classes, including Boolean, Byte, Character, Double, Float, Integer, Long, and Short. These classes are called Wrapper classes and I will provide insight into their use in section 8.3.

```
Elements in the iceCream ArrayList: [Vanilla, Chocolate, Strawberry, Teaberry]
Elements in flavors: [Fudge, Neopolitan]
Ice Cream plus flavors: [Vanilla, Chocolate, Strawberry, Teaberry, Fudge, Neopolitan]
```

Figure 8.4 A screenshot of the output showing a list of ice cream flavors created with an ArrayList

Listing 8.1 Create an ArrayList to hold ice cream flavors

```
1. import java.util.ArrayList;
2. public class Test {
3.     public static void main(String[] args) {
4.         ArrayList<String> iceCream = new ArrayList<String>(); // #A
5.         iceCream.add("Vanilla"); // #B
6.         iceCream.add("Chocolate"); // #B
7.         iceCream.add("Strawberry"); // #B
```

```

8.         iceCream.add("Teaberry");           // #B
9.         System.out.println("Elements in the iceCream ArrayList: "
10.            + iceCream);
11.
12.        ArrayList<String> flavors = new ArrayList<String>();
13.        flavors.add("Fudge");
14.        flavors.add("Neopolitan");
15.        System.out.println("Elements in flavors: " + flavors);
16.
17.        iceCream.addAll(flavors);           // #C
18.        System.out.println("Ice Cream plus flavors: " + iceCream);
19.    }
20. }

```

#A Create a new `ArrayList` to hold the ice cream flavors

#B Use the `add(E e)` method to add four flavors to the list

#C Add the second list to the first list using `addAll(Collection<? extends E> c)`

Quick Check 8-1:

1. Which statement correctly creates an `ArrayList` of size 5:

- a. `ArrayList<String> list = new ArrayList<String>(5);`
- b. `ArrayList<String>(5);`
- c. `ArrayList<String> list = new ArrayList<String>();`

2. Which statement(s) cause an error, given:

```
ArrayList<String> fruit = new ArrayList<String>(5);
```

- a. `fruit.add(5);`
- b. `fruit.add("apple");`
- c. `ArrayList<String> list = new ArrayList<String>(fruit);`
- d. `ArrayList<Double> list2 = new ArrayList<Double>(fruit);`

8.2 Scanner class

In this section, I want to demonstrate how to use the `Scanner` class, which is found in the `java.util` package. When reading input from the user, the `Scanner` class makes it easy. Before reading data from the command line, the first thing is to create a new `Scanner` object, for example:

```
Scanner in = new Scanner(System.in);
```

The `Scanner` class can also be used to read data from a file.

Once the `Scanner` object is created, there are several methods that can be used to parse the input into tokens. For example, the `Scanner` class has a method that can check the next value to determine the type of data. If you are expecting the value to be an integer, there is a method that returns true or false depending on the data. There are methods for all the

primitive data types, even a method that checks for any data at all, `hasNext()`. Each method listed in Table 8.1 returns a value of true or false.

Table 8.1 List of `hasNext` methods for the `Scanner` class

Method
<code>hasNext()</code>
<code>hasNextBoolean()</code>
<code>hasNextByte()</code>
<code>hasNextDouble()</code>
<code>hasNextFloat()</code>
<code>hasNextInt()</code>
<code>hasNextLine()</code>
<code>hasNextLong()</code>
<code>hasNextShort()</code>

Similar methods exist for actually reading the values. It is important to note that the `hasNext` method only check for next token in the scanner's input and does **not** actually remove the data from the input stream. For example, we can use the `hasNextInt()` method to check for an integer. If the value is an integer, then the result is true, but we must then use `nextInt()` to retrieve that value. Listing 8.2 is code that can be used to test that the user entered an integer value.

Listing 8.2 Code used to read in an Integer value

```

1.  import java.util.Scanner;
2.  public class Test {
3.      public static void main(String[] args) {
4.          Scanner in = new Scanner(System.in);
5.          int count = 0;
6.          System.out.println("Enter an integer: ");
7.          while(!in.hasNextInt()) {    // #A
8.              System.out.println("Invalid value, enter an integer");
9.              in.nextLine();        // #B
10.         }
11.         count = in.nextInt();    // #C
12.     }
13. }
```

#A The while loop condition tests if the value is **NOT** an integer

#B If it is not an integer, we need to remove the value from the input buffer

#C Read the valid integer value from the input buffer

Another useful class is the `Object` class. This class can be used to refer to any `Object`, regardless of its original `Class` type. For an example of using the `Object` class, refer to figure 8.5 which shows the sample output from executing the code in listing 8.3.

```
Object value: Wednesday
Object class: class java.lang.String
```

Figure 8.5 Screenshot of the output for code listing 8.3

Listing 8.3 Code that uses the `Object` class

```
1. import java.lang.Object;
2. import java.util.ArrayList;
3. public class Test {
4.     public static void main(String[] args) {
5.         ArrayList<String> list = new ArrayList<String> (7);
6.         list.add("Monday");
7.         list.add("Tuesday");
8.         list.add("Wednesday");
9.         list.add("Thursday");
10.        list.add("Friday");
11.        list.add("Saturday");
12.        list.add("Sunday");
13.        Object obj = list.get(2);
14.        System.out.println("Object value: "+obj.toString() +
15.            "\nObject class: " + obj.getClass());
16.    }
17. }
```

In this example, I used two of the methods from the `Object` class: `toString()` and `getClass()`. Other helpful methods include `clone()` and `equals(Object o)`. Since `Strings` are stored as objects in Java, we cannot use the `==` to compare two `String` objects. So, if we want to compare these two objects, the code must use the `.equals(Object o)` method from the `Object` class to make the comparison. If we tried to use `==`, the program would try to compare the reference value of each object, which contains the location in memory for each object. This is only the same when two objects are pointing to the exact same location, in other words, they are aliases of each other. Also, in this example, I have explicitly imported the `java.lang.Object` class, but it is not required. The `java.lang` package is imported by default into every Java program.

Quick Check 8-2

- Given the code in listing 8.3, what is the result of this statement:

```
System.out.println(obj.equals("Wednesday"));
```

- true
- false

8.3 Wrapper classes

As I stated earlier, arrays can hold lists of primitive data/lists of objects and ArrayLists can hold lists of objects. There are times when it is helpful to create an ArrayList that contains primitive data. For example, we might need an ArrayList that contains `double` values. But, since the ArrayList only works with classes and objects, we have to put the primitive data inside a class first, the wrapper classes were designed just for this task.

8.3.1 Wrapper classes for primitive data types

Each primitive data type has a corresponding wrapper class that allows us to create objects that hold primitive data. Here is a chart of all the primitive data types and their corresponding wrapper classes:

Table 8.2 This table shows the corresponding wrapper class for each primitive data type.

Primitive Data Type	Wrapper Class	Constructors
<code>int</code>	<code>Integer</code>	<code>Integer(int value)</code> <code>Integer(String s)</code>
<code>double</code>	<code>Double</code>	<code>Double(double value)</code> <code>Double(String s)</code>
<code>boolean</code>	<code>Boolean</code>	<code>Boolean(boolean value)</code> <code>Boolean(String s)</code>
<code>char</code>	<code>Character</code>	<code>Character(char value)</code> <code>Character(String s)</code>
<code>float</code>	<code>Float</code>	<code>Float(float value)</code> <code>Float(String s)</code>
<code>short</code>	<code>Short</code>	<code>Short(short value)</code> <code>Short(String s)</code>
<code>byte</code>	<code>Byte</code>	<code>Byte(byte value)</code> <code>Byte(String s)</code>
<code>long</code>	<code>Long</code>	<code>Long(long value)</code> <code>Long(String s)</code>

Notice that each wrapper class starts with a capital letter.

Table 8.2 lists the constructors for each of the wrapper classes. It is easy to create objects from either the corresponding primitive data type or even a String. For example:

```
Integer a = new Integer(5);
```

```
Integer b = new Integer("5");
```

Both statements create Integer objects containing the value 5.

Here is an example, many educational institutions use Learning Management Systems (LMS) to help track students in a class, along with their grades during a semester. When designing the LMS, the developer needs to allow for a course that has a small number of students, but it also has to handle courses with very large numbers of students.

To keep track of the grades for a particular assignment, it makes sense to use an ArrayList so it can grow and shrink as needed. Once the ArrayList is declared, a loop can ask the user to enter grades using -1 as the sentinel value to indicate there are no more grades to be entered.

Listing 8.4

```
1.  ArrayList<Double> grades = new ArrayList<Double>();
2.  double grade = in.nextDouble();
3.  while(value >= 0) {
4.      grades.add(grade);    ##A
5.      System.out.println("Enter next grade: ");
6.      grade = in.nextDouble();
7.  }
```

##A Line 4 shows how the `grade` variable is autoboxed from `double` to `Double` when added to the ArrayList

Autoboxing is the term used to describe the conversion from a primitive data type to its corresponding wrapper class. This occurs automatically by the compiler when a primitive data type is added to an ArrayList. The compiler also unboxes automatically when the values are taken from the ArrayList and put into a primitive data type. Listing 8.4 shows the code for finding the average of a series of grades.

Listing 8.5 Code used to read grades and find the average

```
1.  import java.util.ArrayList;
2.  import java.util.Scanner;
3.  public class Grades {
4.      public static void main(String[] args) {
5.          Scanner in = new Scanner(System.in);
6.          ArrayList<Double> grades = new ArrayList<Double>();    ##A
7.          System.out.println("Enter first grade: ");
8.          double value = in.nextDouble();    ##B
9.          while (value >= 0) {
10.             grades.add(value);    ##C
11.             System.out.println("Enter next grade: ");
12.             value = in.nextDouble();
13.         }
14.         double total = 0, count = 0;
15.         for(double s: grades) {    ##D
16.             total += s;
17.             count++;
18.         }
```

```

19.         System.out.printf("\nThe average scores is: %.2f \n",
20.             (total/count));
21.     }
22. }

```

#A Create an ArrayList to hold the student's grades

#B Read the grade into a variable defined as a double

#C The value is autoboxed from the primitive datatype of double to the class Double

#D An enhanced for loop is used to automatically debox each value from Double to double

8.3.2 Wrapper class methods

Each wrapper class has similar methods available from the API. These methods provide the ability to compare wrapper objects, convert from a primitive data type to an object, parse a String to a numeric object and much more.

Table 8.3 This table shows a subset of the methods for the Integer class.

Method	Example	Description
compare(int x, int y)	Integer.compare(15, 16)	compares x and y, returns: negative value if (x < y) zero if (x = y) positive value if (x > y)
compareTo(Integer x)	Integer x = 15; Integer y = 16; y.compareTo(x)	compares x and y, returns: negative value if (x < y) zero if (x = y) positive value if (x > y)
doubleValue()	Integer x = 15; x.doubleValue()	returns the Integer value as a double
equals(Object obj)	Integer x = 15; Integer y = 16; x.equals(y)	returns true if they are equal, otherwise false
floatValue()	Integer x = 15; x.floatValue()	returns the value of this Integer as a float
intValue()	Integer x = 15; x.intValue()	returns the value of this Integer as an int
parseInt(String s)	Integer.parseInt("510")	returns an Integer parsed from the String
valueOf(int x)	Integer.valueOf(5);	converts int to Integer

<code>valueOf(String s)</code>	<code>Integer.valueOf("543")</code>	returns the String as an Integer
<code>valueOf(String s, int radix)</code>	<code>Integer.valueOf("1111", 2)</code>	converts String to binary, then to Integer

*Static methods

In table 8.3, some of the methods have the class name preceding the method name, such as `Integer.compare(x, y)`. This syntax indicates that the method is associated with the class and not a specific instance of that class.

Non-static methods require an object to act upon. But static classes are independent of any specific instance of the class. The `Math` class is a great example of static methods. The `Math` class has numerous static methods, for example, if we want to find the square root of 160, we don't have to create an object, we can use `Math.sqrt(160)`;

Quick Check 8-3

1. What is wrong with the following statement:

```
a. ArrayList<double> list = new ArrayList<double>();
```

2. Which statement demonstrates autoboxing:

```
ArrayList<Integer> numbers = new ArrayList<Integers>(5):
```

- a. `numbers.add(new Integer(5));`
- b. `numbers.add(5);`
- c. `int x = numbers.get(0);`

3. Which statement demonstrates unboxing:

```
ArrayList<Integer> numbers = new ArrayList<Integers>(5):
```

- a. `numbers.add(new Integer(5));`
- b. `numbers.add(5);`
- c. `int x = numbers.get(0);`

4. Given this code snippet, answer the questions below.

```
double price = 12.50;
String input = "15.70";
Double price2 = Double.parseDouble(input);
Integer cost = 15;
double cost2 = x.doubleValue();
```

5. What is the value returned from this statement: `Double.compare(price,price2)`

- a. negative value
 - b. zero
 - c. positive value
6. What is the value returned from this statement: `Double.compare(price2, cost2)`
- a. negative value
 - b. zero
 - c. positive value
7. Which statement(s) correctly converts price to the Double `dblPrice`?
- a. `Double dblPrice = price.toDouble()`
 - b. `Double dblPrice = Double.valueOf(price)`
 - c. `Double dblPrice = price;`

8.4 Summary

In this lesson, you learned:

- How to read the Java Standard API documentation
- To navigate the Java API documentation
- How to use classes, interfaces and methods included in the Java API

As you can see from the examples in this lesson, the Java API contains many useful classes and interfaces. As we continue to discuss object-oriented topics in Java, we will rely heavily on the code made available by the Standard Java API.

This lesson also introduced the topic of Wrapper classes. A wrapper class converts a primitive data type to an object. Each primitive data type has a corresponding wrapper class. There are several ways to convert between the object and the primitive value, the easiest is autoboxing and unboxing since this is done automatically by the compiler. But there are several methods that can be helpful especially if you want to convert a number inside a String.

In the next lesson, I will explore the String and StringBuilder classes.

Try this:

Given the information for the `Arrays` class in figure 8.6, which is in the `Java.util` package, write code that creates two integer arrays and then compares the arrays to determine if they are equal. Instead of looping through each element, use the documentation provided to use the `equals(int[] a, int[] b)` method to compare the two arrays.

static boolean	equals (boolean[] a, boolean[] a2) Returns true if the two specified arrays of booleans are <i>equal</i> to one another.
static boolean	equals (byte[] a, byte[] a2) Returns true if the two specified arrays of bytes are <i>equal</i> to one another.
static boolean	equals (char[] a, char[] a2) Returns true if the two specified arrays of chars are <i>equal</i> to one another.
static boolean	equals (double[] a, double[] a2) Returns true if the two specified arrays of doubles are <i>equal</i> to one another.
static boolean	equals (float[] a, float[] a2) Returns true if the two specified arrays of floats are <i>equal</i> to one another.
static boolean	equals (int[] a, int[] a2) Returns true if the two specified arrays of ints are <i>equal</i> to one another.
static boolean	equals (long[] a, long[] a2) Returns true if the two specified arrays of longs are <i>equal</i> to one another.
static boolean	equals (Object[] a, Object[] a2) Returns true if the two specified arrays of Objects are <i>equal</i> to one another.
static boolean	equals (short[] a, short[] a2) Returns true if the two specified arrays of shorts are <i>equal</i> to one another.
static void	fill (boolean[] a, boolean val) Assigns the specified boolean value to each element of the specified array of booleans.
static void	fill (boolean[] a, int fromIndex, int toIndex, boolean val) Assigns the specified boolean value to each element of the specified range of the specified array of booleans.

Figure 8.6 Screenshot of Java API documentation for the `Arrays` class

Note: Other helpful methods in the `Arrays` class include:

- `toString(datatype[] a)` for printing arrays
- `copyOf(datatype[] a, newLength)` makes a copy of the current array, creating a new array of size `newLength`
- `fill(datatype[] a, datatype value)` assigns the specified value to each element in the array

Quick Check 8-1 Solution:

1. Which statement correctly creates an `ArrayList` of size 5:

- a. `ArrayList<String> list = new ArrayList<String>(5);`
- b. `ArrayList<String>(5);`
- c. `ArrayList<String> list = new ArrayList<String>();`

2. Which statement(s) causes an error, given:

```
ArrayList<String> fruit = new ArrayList<String>(5);
```

- a. `fruit.add(5);` (out of bounds)
- b. `fruit.add("apple");`
- c. `ArrayList<String> list = new ArrayList<String>(fruit);`
- d. `ArrayList<Double> list2 = new ArrayList<Double>(fruit);`
(cannot convert String to Double)

Quick Check 8-2:

1. Give the code in listing 8.3, what is the result of this statement:

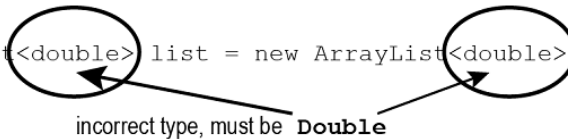
```
System.out.println(obj.equals("Wednesday"));
```

- a. true
- b. false

Quick Check 8-3:

1. What is wrong with the following statement:

a) `ArrayList<double> list = new ArrayList<double>();`



incorrect type, must be **Double**

2. Which statement demonstrates autoboxing:

```
ArrayList<Integer> numbers = new ArrayList<Integers>(5);
```

- a. `numbers.add(new Integer(5));`
- b. `numbers.add(5);`
- c. `int x = numbers.get(0);`

3. Which statement demonstrates unboxing:

```
ArrayList<Integer> numbers = new ArrayList<Integers>(5);
```

- a. `numbers.add(new Integer(5));`
- b. `numbers.add(5);`
- c. `int x = numbers.get(0);`

4. Given this code snippet, answer the questions below.

```
double price = 12.50;
String input = "15.70";
Double price2 = Double.parseDouble(input);
Integer cost = 15;
double cost2 = x.doubleValue();
```

5. What is the value returned from this statement: `Double.compare(price,price2)`

- a. negative value
 - b. zero
 - c. positive value
6. What is the value returned from this statement: `Double.compare(price2, cost2)`
- a. negative value
 - b. zero
 - c. positive value
7. Which statement(s) correctly converts price to the Double `dblPrice`?
- a. `Double dblPrice = price.toDouble();`
 - b. `Double dblPrice = Double.valueOf(price)`
 - c. `Double dblPrice = price;`

Solution to Try This Exercise:

```
The two arrays are not equal!
First array: [1, 2, 3, 4, 5, 6, 7]
Second array: [2, 3, 4, 5, 6, 7, 8]
```

Figure 8.7 Screenshot of output from executing the code in listing 8.4

Listing 8.4 Use the .equals method to check if two arrays contain the same values

```
1. import java.util.Arrays;
2. public class Test {
3.     public static void main(String[] args) {
4.         int[] numbers = {1, 2, 3, 4, 5, 6, 7};
5.         int[] numbers2 = {2, 3, 4, 5, 6, 7, 8};
6.         if(Arrays.equals(numbers, numbers2)) // #A
7.             System.out.println("The two arrays are equal! ");
8.         else
9.             System.out.println("The two arrays are not equal! ");
10.        System.out.println("First array: " + Arrays.toString(numbers)); // #B
11.        System.out.println("Second array: " + Arrays.toString(numbers2));
12.    }
13. }
```

#A Compare two integer arrays using the method from the Arrays class

#B Use the toString method from the Arrays class to print all values of each array

9

String and StringBuilder Classes

After reading lesson 9, you will be able to:

- Work with strings
- Convert from numbers to strings and vice versa
- Choose the correct String method (toLowerCase, toUpperCase, etc.)
- Use the StringBuilder class

This lesson introduces the topic of strings in Java. In Java, strings are objects. A string represents a sequence of individual characters, such as a word or sentence. The String class is widely used in Java and has over sixty methods and thirteen constructors. In addition to the String class, Java also has a StringBuilder class.

Consider This

I'm sure you have been on the receiving end of a piece of mail generated by a marketing firm. Although my physical mail (sometimes referred to as snail mail) has decreased over the years, I still receive solicitations weekly for a new credit card, or maybe even a home equity loan. These mailings have become more personalized over the years, including my name in the actual letter.

This is an example of how a program can be used to merge text. Many programs need to process text and often navigate a large amount of text looking for specific data. For marketing, the company usually has a large list of names and addresses that is the input to a program. This information has to be separated into first name, last name, address line 1, optionally an address line 2, etc. As you think about this process can you think of other types of text that appears on things we own? How about the information printed on a cereal box? Or even the information in this text book. These are all examples of programs that needed to process text and in Java that means they need to read and manipulate strings.

9.1 Working with strings

It is easy to create a variable that holds a string:

```
String groceryItem = "apples";
```

In this example, the grocery item value is considered a *string literal*, which is a sequence of characters enclosed in double quotes. Behind the scenes, the literal is stored as an array of characters and the variable name contains the reference value to find that array in memory.

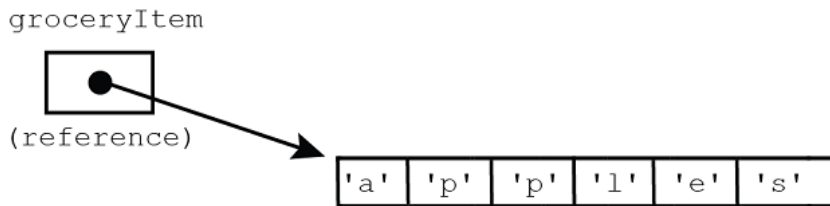


Figure 9.1 Diagram of a string saved as individual characters with a reference variable that points to the start of the array

NOTE:

It is important to note that String literals are immutable (cannot be changed). Java stores string literals in a *string pool*. All String literals are enclosed in double quotes.

It is often helpful to find the length of a string object. The String class has a method, `.length()`, which returns an integer value representing the number of characters in a string, including white space characters such as the end line symbol ("`\n`"). The code snippet below, finds the length of a string entered by the user, then prints out the length:

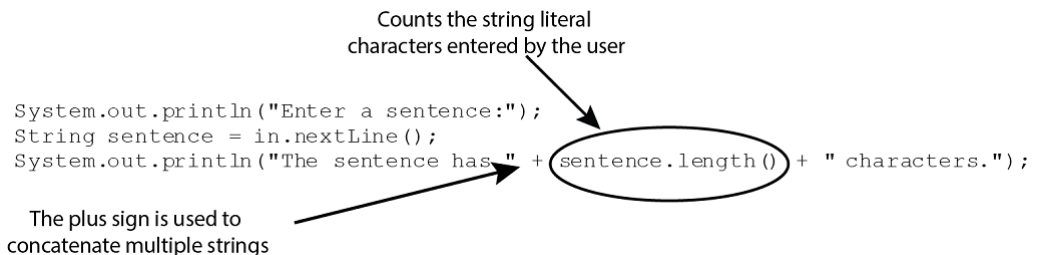


Figure 9.2 Shows how to use the `.length` method and how to combine strings

Notice the plus sign in the `println` statement. In this example, when a string is combined with an integer, it automatically converts the integer value as another string and puts them together. Here we have a string, a number, then another string. All three values are printed as one string. For example, if the user enters the sentence "Today is Wednesday", the output would be:

```
The sentence has 18 characters.
```

There is also a method that can be used to concatenate (or put together) two strings, it is `.concat()`. Here are some examples of using this method:

```
String s1 = "apples";
String s2 = "oranges";
String s3 = s1.concat(" and ").concat(s2);
System.out.println(s3);
```

The result would be: `apples and oranges`

In the example, I had to create a new variable to hold the result of combining `s1` and `s2` (notice in this example I am actually chaining two calls to the `concat` method to create the final version of the `String` which is "apples and oranges"). Remember, strings cannot be changed directly in Java, they are immutable. In the next section, I will review formatting numbers in a string.

Quick Check 9-1:

Given: `String sentence = "Today is Thursday!";`

1. What is the result of this statement: `System.out.println(sentence.length());`
 - a. 19
 - b. 18
 - c. 17
 - d. 20

9.2 Converting between numbers and strings

There are several ways to convert a number to a string. The simplest approach is to append the number to a string, such as: `String age = "" + 21;` Now, the variable `age` contains a string with the value "21". In this example, I am using an empty string value of "", which does not have any spaces between the set of double quotes.

NOTE: In Java, the word `null` is actually reserved as a special value that is recognized by the compiler to indicate that the object is only a reference and does not actually contain any value. Consider it a reference to nothing. In my example, it allows me to create a new string variable without adding any extra characters. The null value is used often in object-oriented programming. It provides an easy test for an object to see if it has been instantiated. Remember our `Car` class from lesson 2? If we create a `Car` object, but don't use the `new` keyword and don't initialize it in any way, it creates a null object, `Car car1;` Right now, `car1` is considered a null object.

In Lesson 8 we learned about the wrapper classes that exist for all of the primitive data types. Each numeric wrapper class has a `valueOf()` method that retrieves the numeric value from a string. This is helpful when parsing string data to find the numeric values in the `String`. Let's look at a program that starts by asking the user to enter a list of numbers separated by

commas. Using that list, the program needs to find the average and then print all the values as integers.

In order to separate the numbers from the commas, I start with the String class method `.split()`. This method uses a delimiter to split the string, in this case it will split it every time it encounters a comma followed by a space. The split method removes the specified value and returns an array of string values. Once the array is created, I can use the wrapper class method, `valueOf()`, to convert the string to a number. Then, it is easy to process the Integer ArrayList and find the average.

```

Enter a list of integers separated by commas and spaces, ie: 1, 2, 3, etc.
5, 17, 23, 15, 94, 56, 74, 50
The average of the numbers is: 41.75
5
17
23
15
94
56
74
50
  
```

Annotations in the screenshot:

- An arrow labeled "read in as a String" points to the input line: "Enter a list of integers separated by commas and spaces, ie: 1, 2, 3, etc."
- An arrow labeled "rounded to two decimal points" points to the value "41.75" in the output line "The average of the numbers is: 41.75".

Figure 9.3 Screen shot of the output from code in Listing 9.1

Listing 9.1 This program converts from string to numeric

```

1. import java.util.ArrayList;
2. import java.util.Scanner;
3. public class Test {
4.     public static void main(String[] args) {
5.         ArrayList<Integer> arrList = new ArrayList<Integer>();
6.         Scanner in = new Scanner(System.in);
7.         double total = 0, average = 0;
8.         System.out.println("Enter a list of integers separated by commas and
spaces, " +
9.             "ie: 1, 2, 3, etc.");
10.        String input = in.nextLine();
11.        String[] strNumbers = input.split(", ");           // #A
12.        for(int i = 0; i<strNumbers.length; i++) {
13.            arrList.add(Integer.valueOf(strNumbers[i])); // #B
14.            total += arrList.get(i);
15.        }
16.        average = total/arrList.size();
17.        System.out.printf("The average of the numbers is: %.2f\n", average);
    // #C
18.        for(int j = 0; j < strNumbers.length; j++) {
19.            System.out.println(strNumbers[j]);
20.        }
21.    }
22. }
  
```

#A The `String` method, `split()`, is used for separating strings into parts

#B Use the `valueOf()` method to convert the strings to integers

#C Use formatting to print the average to only two decimal points

In this example, I demonstrated how to split a string into smaller strings and convert strings to numeric values so they can be used in a mathematical computation.

Quick Check 9-2:

1. Given a string, `input`, that contains a series of numbers separated by semicolons, how can you parse that string:
 - a. `input.split(";")`
 - b. `input.parseString(";")`

9.3 String methods

The `String` class is part of the `java.lang` package in the Java API. Therefore, you can find all the documentation including a complete list of all available methods from the Oracle Help Center website (<https://docs.oracle.com/en/>). Here is a list of some other helpful and commonly used methods:

Table 9.1 Commonly used `String` methods

Method	Example	Description
<code>charAt(int index)</code>	<code>word.charAt(0)</code>	returns the char value at index
<code>compareTo(String str)</code>	<code>word.compareTo(word2)</code>	compares <code>x</code> and <code>y</code> , returns: negative value if (<code>x < y</code>) zero if (<code>x = y</code>) positive value if (<code>x > y</code>)
<code>concat(String str)</code>	<code>word.concat(word2)</code>	adds <code>word2</code> to the end of <code>word</code>
<code>contains(CharSequence s)</code>	<code>word.contains("a")</code>	returns true if value is found, otherwise returns false
<code>equals(Object obj)</code>	<code>word.equals("yes")</code>	returns true if strings are equal, otherwise false
<code>equalsIgnoreCase(String str)</code>	<code>word.equalsIgnoreCase("no")</code>	returns true if strings are equal regardless of capitalization, otherwise false
<code>length()</code>	<code>word.length()</code>	returns the number of characters in the string

<code>split(String regex)</code>	<code>word.split(", ")</code>	splits the string before and after each match of the regex value
<code>substring(int index)</code>	<code>word.substring(4)</code>	returns a substring starting at the index value until the end
<code>substring(int begin, int end)</code>	<code>word.substring(0, 4)</code>	returns a substring starting at the index value of begin to end-1
<code>toLowerCase()</code>	<code>word.toLowerCase()</code>	returns the string in all lower-case letters
<code>toUpperCase()</code>	<code>word.toUpperCase()</code>	returns the string in all upper-case letters

When working with strings, it is often helpful to access individual characters in the string. A combination of the `.length()` and `.charAt(int index)` methods can be used together to access each individual character in a string. Here is a code snippet that prints each character in reverse order:

```
for(int i = sentence.length() - 1; i >= 0; i--) {
    System.out.print(sentence.charAt(i));
}
```

It is important to remember that the characters in a string start at index 0, so the last character is located at `length - 1`. If the program tries to access the character at the index `length`, it will be out of bounds. In this code snippet, I started the index value at the last character and subtract one each time. Once the value gets to `-1` it will stop.

For example, if the program needs to ask the user for a response such as "yes" or "no", it might help to convert the string to either all upper case or lower case first. That way if the user enters "Yes" or "yes", we know they are the same answer. Here is a simple code snippet that converts the user response to all lower case before we check to see if the value matches "yes" or "no":

```
response = in.next().toLowerCase();
if(response.equals("yes")) System.out.println("You entered yes");
```

Another option is to use the comparison method that ignores upper and lower case letters and just checks to see if the words match:

```
if(response.equalsIgnoreCase("yes")).
```

Before moving on to the next section, I want to review how to use the substring method. This method returns part of the original string. If only one index value is included in the argument list, then it will start at that value and return everything from there until the end.

If the substring method has two values, then it starts at the first index value and stops at one less than the second value. Here are some examples, given the original string: `String name = "Peggy Fisher";`

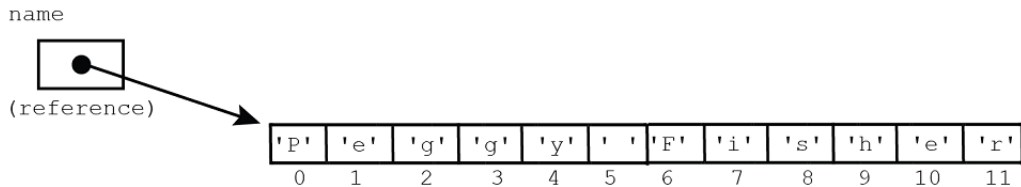


Figure 9.2 Diagram of the string variable name showing the index values of each character

Here is the code to retrieve the first name and last name using the substring method:

```
String firstName = name.substring(0,5);
String lastName = name.substring(6);
```

Notice that the first statement starts at position zero but stops at one position prior to the space so that the entire first name is now in the variable `firstName`. The next statement starts at position 6 and goes to the end, so it now has the string "Fisher".

Quick Check 9-3:

Given: String sentence = "Today is Monday, what a great day to program in Java!!!";

1. What is the result of this code snippet:

```
for(int k = 0; k <= sentence.length(); k++)
    System.out.println(sentence.substring(24));
```

- a. "great day to program in Java!!!"
- b. " great day to program in Java!!!"
- c. "a great day to program in Java!!!"
- d. "day to program in Java!!!"

9.4 StringBuilder class and methods

The big benefit of using the `StringBuilder` class is that it allows us to edit a string, remember, strings are immutable, so using the `StringBuilder` class helps to create a string that can be changed. This class is helpful if we need to append strings together, insert characters into a string, and so on. If we use the `String` class for these actions, each time the code would be creating a new `String` object. This can quickly use up a lot of memory which could increase our run time depending on how many strings are being updated.

The alternative is to use a `StringBuilder` object when we know that the `String` object might need to be updated in one of these ways: appending strings, inserting strings, replacing characters or string subsets, reversing a string, deleting all or part of a string.

To explain the methods that facilitate these actions in the `StringBuilder` class, Listing 9.2 shows a complete program that uses some of these methods. This program asks the user to enter a word or phrase, then checks to see if the string is considered a palindrome. A palindrome is a word or sentence that is spelled the same way forward and in reverse, such as `racecar`, `civic`, `level`, `kayak`, `noon`, and so on.

Notice that I do not need to import the `StringBuilder` class, it is also part of the `java.lang` package which is automatically imported for all Java programs.

Listing 9.2 This program tests a string to see if it is a palindrome

```

1.  import java.util.Scanner;
2.  public class FunWithStrings {
3.      public static void main(String[] args) {
4.          Scanner in = new Scanner(System.in);
5.          boolean isPalindrome = true;
6.          StringBuilder text = new StringBuilder();    //#A
7.          System.out.print("Enter a sentence/word and I will test to see if it is a"
+
8.              + " palindrome: ");
9.          String input = in.nextLine();
10.         text.append(input);                          //#B
11.         StringBuilder textReversed = new StringBuilder();    //#C
12.         textReversed.append(input).reverse();          //#C
13.         for (int i = 0; i < input.length(); i++) {      //#E
14.             if (text.charAt(i) != textReversed.charAt(i)) {    //#E
15.                 isPalindrome = false;                    //#E
16.                 break;                                     //#E
17.             }                                              //#E
18.         }                                               //#E
19.         System.out.println("original: " + text + " length: "+text.length());
20.         System.out.println("reversed: " + textReversed);
21.         if (isPalindrome) {
22.             System.out.println(text + " is a palindrome! ");
23.         }
24.         else
25.             System.out.println(text + " is not a palindrome, try again :-) ");
26.     }
27. }

```

#A Create an empty `StringBuilder` object, `text`

#B Add the word or sentence entered by the user to the variable `text` (using `append`)

#C Create a new `StringBuilder` object, `textReversed`, that contains the original value entered by the user in reverse order

#D Compare each character in the original word/phrase to the reversed word/phrase, if they are all the same, it is a palindrome

In Listing 9.2, the code starts by creating a new `StringBuilder` object, `text`. Then using the `.append` method from the `StringBuilder` class, the word/phrase entered by the user is added to

the empty object. To test if this is a palindrome, the program creates a second object that contains the same value in reverse. Notice that line 11 creates a second `StringBuilder` object that is then assigned the value of the original string in reverse order using the methods on line 12. Now it is easy to test if this is a palindrome but just comparing the two strings character for character to see if they are the same.

NOTE: since `StringBuilder` objects have a `toString()` method, it is possible to make this code even shorter by using this `if` statement:

```
if(text.toString().equals(textReversed.toString()))
    System.out.println(text + " is a Palindrome!!");
```

The `StringBuilder` can be used when we need to combine two strings without creating a third string. Consider a program that starts by asking the user for their name and then prints out a greeting using their name. What `StringBuilder` methods should this program use? Listing 9.3 shows one version of how this program can be written:

Listing 9.3 This program prints a greeting using the user name and time of day

```
1. import java.util.Scanner;
2. public class Greeting {
3.     public static void main(String[] args) {
4.         Scanner in = new Scanner(System.in);
5.         StringBuilder greeting = new StringBuilder();    ##A
6.         System.out.println("Enter your name: ");
7.         String name = in.nextLine();
8.         greeting.append("Hello, ");                    ##B
9.         greeting.append(name);                          ##C
10.        System.out.println(greeting);
11.    }
12. }
```

#A Create an empty `StringBuilder` object, `greeting`

#B Add the word "Hello," to the greeting using the `append` method

#C Add the name entered by the user to the end of the greeting

Now, let's add a step to insert a value into our greeting. To keep things simple, we can add "and welcome" to the message, so it will say "Hello, and welcome". See the revised code listing 9.4:

Listing 9.3 This program prints a greeting using the user name and time of day

```
1. import java.util.Scanner;
2. public class Greeting {
3.     public static void main(String[] args) {
4.         Scanner in = new Scanner(System.in);
5.         StringBuilder greeting = new StringBuilder();
6.         System.out.println("Enter your name: ");
7.         String name = in.nextLine();
8.         greeting.append("Hello, ");
```

```

9.         greeting.append(name);
10.        greeting.insert(6, "and Welcome,"); // #A
11.        System.out.println(greeting);
12.    }
13. }

```

#A Insert the words "and Welcome, " into the `StringBuilder` object `greeting`

The `StringBuilder` class also offers several other methods that can be used to retrieve a single character using `.charAt(index)`, or delete a character or sequence of characters. It can even find the first occurrence of a character and return the index value of the character. Check out the [JavaDoc](https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html) for more information about this important class. (<https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html>)

Quick Check 9-4:

- Which statement correctly prints each character on a separate line, given this code:

```
StringBuilder quote = "It is often easier to ask for forgiveness than to ask for permission";
```

- `for(int i = 0; i < greeting.size; i++) {System.out.println(greeting.charAt(i));}`
 - `for(int i = 0; i < greeting.length; i++) {System.out.println(greeting.charAt(i));}`
 - `for(int i = 0; i < greeting.length(); i++) {System.out.println(greeting.charAt(i));}`
 - `for(int i = 0; i < greeting.length(); i++) {System.out.println(greeting.delete(i));}`
- Which is **not** a method for the `StringBuilder` class?
 - `substring(int x)`
 - `equalsIgnoreCase(Object obj)`
 - `toString()`

9.5 Summary

In this lesson, you learned:

- How to work with strings
- How to convert from numbers to strings and vice versa
- To choose the correct String method (`toLowerCase`, `toUpperCase`, etc.)
- How to compare strings
- When to use the `StringBuilder` class

This lesson reviewed the use of the `String` and `StringBuilder` classes in Java. When programming in general it is often necessary to read and manipulate text either from the user or from a file. Remember, a string can be converted to a `StringBuilder` using a constructor and a string builder object can be converted to a string using the `toString()` method.

In the next lesson, I will introduce the topic of the static classes and methods.

TRY THIS: Write code that reads in a list of first names separated by commas and store them in a `String` array. Then use the `StringBuilder` class to print a greeting for each name, such as "Hello, Peggy. How are you today?". Start by creating a new `StringBuilder` object that contains "Hello, . How are you today?", then insert the name and print the result. (remember to delete the name from the `StringBuilder` object each time, so the previous name does not get included in the output). Here is some sample output from the program:

```
Enter a series of names separated by commas:
Peggy, Bob, Cy, Harley
Hello, Peggy. How are you today?
Hello, Bob. How are you today?
Hello, Cy. How are you today?
Hello, Harley. How are you today?
```

Figure 9.5 This is sample output from the Try this activity.

Quick Check 9-1 Solution:

Given: `String sentence = "Today is Thursday!";`

1. What is the result of this statement: `System.out.println(sentence.length());`
 - a. 19
 - b. 18**
 - c. 17
 - d. 20

Quick Check 9-2 Solution:

1. Given a string, `input`, that contains a series of numbers separated by semicolons, how can you break up the string into just numbers:
 - a. `input.split(";")`
 - b. `input.parseString(";")`

Quick Check 9-3 Solution:

Given: `String sentence = "Today is Monday, what a great day to program in Java!!!";`

1. What is the result of this code snippet:

```
for(int k = 0; k <= sentence.length(); k++)
System.out.println(sentence.substring(24));
```

- a. "great day to program in Java!!!"
- b. " great day to program in Java!!!"
- c. "a great day to program in Java!!!"
- d. "day to program in Java!!!"

Quick Check 9-4 Solution:

1. Which statement correctly prints each character on a separate line, given this code:

```
StringBuilder quote = "It is often easier to ask for forgiveness than to ask for permission";
```

- a. `for(int i = 0;i<greeting.size; i++) {System.out.println(greeting.charAt(i));}`
- b. `for(int i = 0;i<greeting.length; i++) {System.out.println(greeting.charAt(i));}`
- c. `for(int i = 0;i<greeting.length(); i++) {System.out.println(greeting.charAt(i));}`
- d. `for(int i = 0;i<greeting.length(); i++) {System.out.println(greeting.delete(i));}`

2. Which is not a method for the `StringBuilder` class?

- a. `substring(int x)`
- b. `equalsIgnoreCase(Object obj)`
- c. `toString()`

Solution to the Try This activity:**Listing 9.4 This code asks for a list of names, then uses a `StringBuilder` to prints personalized greetings**

```
1. import java.util.Scanner;
2. public class Greeting {
3.     public static void main(String[] args) {
4.         Scanner in = new Scanner(System.in);
5.         StringBuilder greeting = new StringBuilder();
6.         System.out.println("Enter a series of names separated by commas:");
7.         String names = in.nextLine();
8.         String[] name = names.split(", ");           // #A
9.         greeting.append("Hello, . How are you today?"); // #B
10.        for(int i = 0;i<name.length;i++) {
11.            greeting.insert(7, name[i]);           // #C
12.            System.out.println(greeting);         // #C
13.            greeting.delete(7,(7+name[i].length())); // #D
14.        }
15.    }
16. }
```

#A Use the `split` method to create separate strings for each name using the comma as a delimiter

#B Add the greeting to the `StringBuilder` object using the `append` method

#C Insert the name to personalize the greeting in the `StringBuilder` object, print the message

#D Remove the inserted name to prepare for the next name

10

Static Methods and Variables

After reading lesson 10, you will be able to:

- Differentiate between static methods and instance methods
- Understand when to use the static keyword
- Correctly define static methods and static variables

This lesson introduces the topic of the **static** keyword in Java. In Java, using the keyword **static** indicates that the method is a **class** method or the variable is a **class** variable as opposed to an instance method or instance variable. Every Java application uses the static keyword at least once for the main method:

```
public static void main(String[] args) {...}
```

In this lesson I will review the impact of using the static keyword for both methods and variables. In addition, since the keyword **final** is often used when creating static variables, this lesson reviews the impact of defining static final variables.

Consider This

Recently a friend of yours decided to start selling t-shirts online with custom graphics. You are working with them to create a Java application to keep track of the inventory. For this example, let's concentrate on the inventory of shirts only (I'm sure there are other supplies needed for the business such as ink, screenprints, etc.). The project starts by creating a TShirt class with instance data about the size of the shirt, color, graphic, and price.

When we create TShirt objects, they each have distinct copies of the instance variables that are not shared from one object to the other. It might be helpful to have some variables shared with all the TShirt objects. For example, maybe your friend wants to include an item number for each shirt, starting at 100 and then incrementing the value each time a new shirt is created. We need to add the item number field to our instance data, but we also need to keep track of the next available item number for the next shirt.

The next item number variable is not associated with a specific shirt, but it is used to get the next number and then it is increased by one each time. So, we need to add a variable to our instance data for the item number and a second static variable, next item number, to the class which starts at 100. This variable must be declared as static so the compiler knows there is only one copy of this value.

10.1 Static Methods

The first thing to know about static methods is that they exist independently of any instances of a class. They are actually shared across all instances of the class. It is easy to find examples of static methods, many of the classes that are available in the Java API contain static methods. The Math class has numerous methods that can be used without first creating a Math object. So, they are not directly tied to an instance of the Math class. Instead, the class name is used with the dot notation to call the method. To find the max of two integers, we can use a method from the Math class:

```
int biggest = Math.max(4, 76);
```

This statement returns the value 76 and places that value in the variable `biggest`. Since I don't need an object, I must use the class name with the dot and the method name. The method does not act upon a specific object.

As I stated in the lesson introduction, every Java application has at least one static method, the main method. The main method is declared as static since there can only be one instance of the main method for each application. The Java virtual machine looks for the specific method signature: `public static void main(String[] args)` to find the starting point of any application. Java applications can have one or many separate files that work together to form the application. But there can only be one main method, so by declaring it static, it ensures there is only one. If you try to execute code that does not include a static main method, you get an error message: "Error: Could not find or load main class ...".

There are rules about whether a static method can call another static method, or can a static method call an instance method. Table 10.1 shows the rules for static methods and variables:

10.1 Rules for using static methods and static variables

instance method	can access instance variables	can access instance methods directly
instance method	can access class (static) variables	can access class (static) methods directly
Class methods	can access class (static) variables	can access class (static) methods directly
Class methods	cannot access instance variables	cannot access instance methods directly, they must use an object reference

NOTE: class methods **cannot** use the **this** keyword as there is no instance for this to refer to.

Listing 10.1 is a code snippet that shows an example of a class that contains three methods and six variables. Two of the methods are static and the third method is an instance method (any method without the keyword `static` is considered an instance method). The static methods are: `assignNextNumber` and `main`. The `print` method is an instance method since it does not include the keyword `static`.

10.1 Example of a class with both static and non-static methods

```

1.  public class TShirt {
2.      static int nextItemNumber = 100;    //A
3.      int itemNumber;
4.      String size, color, graphic;
5.      double price = 14.99;
6.      static int assignNextNumber() { return nextItemNumber++; } //B
7.      void print() {System.out.println("TShirt number: "+itemNumber);} //C
8.      public static void main(String[] args) {
9.          TShirt tee1 = new TShirt();    //D
10.         tee1.color="pink";            //E
11.         tee1.graphic="Duke";          //E
12.         tee1.itemNumber = assignNextNumber(); //F
13.         print();                       //G
14.     }
15. }

```

#A Static variable `nextItemNumber` is used to keep track of the next item number

#B This is a static method used to assign the next item number, then increase it by one

#C This is an instance method that prints out the shirt item number

#D Create a new tshirt object

#E Provide values for the instance data

#F Use the static method to get the next available item number

#G Print the tshirt item number

In this example, line 13 causes a compile time error, this occurs because we cannot call a non-static method from a static method. In this case, I'm trying to call the non-static method `print` from the static `main` method. To fix this error, it is possible to use the `TShirt` object created on line 9 to call the instance method and print the item number for this t-shirt:

```
tee1.print();
```

Since `print` is an instance method, we must access it through an object that is created using the class `TShirt`.

Figure 10.1 shows the variables and methods that are static on the left. The right is a list of instance variables and methods. There can only be one variable for `nextItemNumber` and one `main` method. All `TShirt` objects are given copies of the instance data and the price is assigned to 14.99 by default. This can be changed once an object is created. But if most of the t-shirts cost 14.99, this saves time by automatically assigning this value.

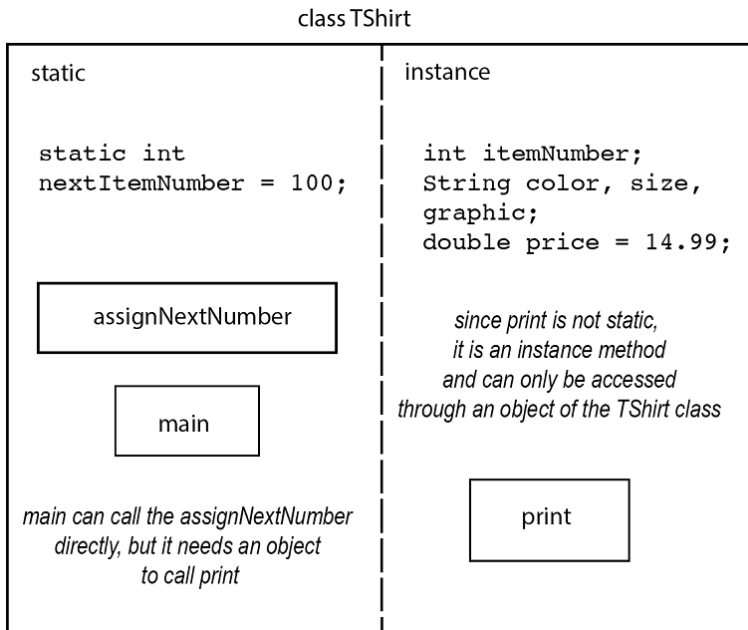


Figure 10.1 This diagram depicts the access of the static and instance methods in the class TShirt

To decide if a method should be static, consider the question, does the program need to call this method independent of any objects? If the answer is yes, then it should be static. If the method only makes sense when attached to an object, then it should be an instance method.

Quick Check 10-1:

Given this code snippet, answer the following questions:

```
public class Vehicle {
    private String make, model, color, vin;
    private int year;

    // add a class variable for the
    // number of Vehicle objects instantiated
    private static int numberOfVehicles = 0;
    public void getVin() {return vin;}
    public static double calculateMPG(double miles, double gallons) {
        return miles/gallons; }
    public void print() { System.out.println(make + ", " + model);
        ...
    public static void main(String[] args) {
        Vehicle veh = new Vehicle();
    }
}
```

1. Which statement correctly updates the numberOfVehicles?

- a. `veh.numberOfVehicles = veh.numberOfVehicles + 1;`
- b. `numberOfVehicles += 1;`
- c. `veh.numberOfVehicles += 1;`

2. Which statement correctly obtains the vin number for vehicle veh?

- a. `veh.getVin();`
- b. `getVin();`
- c. `veh.getVin(String make);`

3. Which statement correctly prints the make and model of a vehicle?

- a. `veh.print();`
- b. `print(String vin);`
- c. `print();`

10.2 Static Variables

Static variables are often used to help with memory management. Consider the situation where we have variables that contain values that are shared across multiple objects. By defining the variable as static, only one copy is required for the duration of the program execution. Since there is only one copy, memory is allocated once when the class is loaded in memory. This is different from the instance variables for a class which have their own copies of all instance variables.

Static variables are useful when the information is the same for all instances of a class. For example, if we had an Employee class and all employees worked for LinkedIn, we could create a static variable for the company name in the Employee class that allocates space for that variable once, even if we have 1,000 employees. A static variable is shared across all instances of a class. Here is an example:

```
class Employee {
    String name;
    static String company = "LinkedIn";
    //other demographic information
}
```

Let's revisit the TShirt class from the prior section to review the static variable `nextItemNumber`.

10.2 Example of a class with both static and non-static methods

```
1. public class TShirt {
2.     static int nextItemNumber = 100;        // #A
3.     int itemNumber;
4.     String size, color, graphic;
5.     double price = 14.99;
6.     static int assignNextNumber() { return nextItemNumber++; } // #B
```

```

7.     void print() {System.out.println("TShirt number: "+itemNumber);}
8.     public static void main(String[] args) {
9.         TShirt tee1 = new TShirt();
10.        tee1.color="pink";
11.        tee1.graphic="Duke";
12.        tee1.itemNumber = assignNextNumber();  // #C
13.        tee1.print();
14.    }
15. }

```

#A Static variable `nextItemNumber` is used to keep track of the next item number

#B This is a static method used to assign the next item number, then increase it by one

#C Print the tshirt item number using the dot notation since print is an instance method

Static variables are often used to either keep a running total of all objects or to provide values that are not dependent on a specific object. In our TShirt class, a variable is defined to hold the next item number. Each object has a unique item number. On line 2, I have declared a static variable called `nextItemNumber` and assigned it a starting value of 100. Since this variable is considered static, there is only one copy of this variable that is shared by all the TShirt objects. This number gets updated every time the static method `assignNextNumber()` is called. Notice that this method first returns the current number so that the calling TShirt object is assigned the next number, then the `++` is executed and one is added to the number so it is ready for the next TShirt.

NOTE: The `++` operator adds one to the variable. It can be located before the variable, in which case it adds one before completing the statement. If it is located after the variable, it adds one after the statement has executed. In this case, the variable is updated after it is returned to the calling program so that it has the next number is stored and ready for the next TShirt object.

If you have ever opened a bank account, you know the bank assigns each customer a unique bank account number. In order to identify the next available account number, the bank needs a static variable in addition to the account number associated with the bank account object. If I open an account today, I might get a bank account number of 1234. The next customer should get account number 1235, and so on. Each customer has a unique number, but the bank account class must also have a static variable with the most recently assigned account number that is updated every time a new account is opened.

Listing 10.2 contains another example, an Astronaut class that includes static and instance variables.

10.3 Example of a class with a static variable

```

1.     public class Astronaut {
2.         private static String companyName = "NASA";           // #A
3.         private String fName, lName;
4.         private int id;
5.         private static int nextIDNumber = 1;
6.         public String getName() {return fName + " " + lName;}

```

```

7.     public void print() {System.out.println("Astronaut: " + fName + " " +
8.         lName);}
9.     public static void main(String[] args) {
10.        Astronaut astro1 = new Astronaut();
11.        astro1.fName = "Howard";
12.        astro1.lName = "Wolowitz";
13.        astro1.id = nextIDNumber;
14.        nextIDNumber++;
15.        Astronaut astro2 = new Astronaut();
16.        astro2.fName = "John";
17.        astro2.lName = "Glenn";
18.        astro2.id = nextIDNumber;
19.        nextIDNumber++;
20.        System.out.println(companyName + "\n=====");    // #B
21.        astro1.print();
22.        astro2.print();
23.    }
24. }
25.

```

#A This statement defines a static variable

#B Print the companyName

The variable for the company name is defined as static since all the astronauts in our example work for NASA (Although Howard Wolowitz is actually a character from the Big Bang Theory). The class also includes a static variable that is used to get the next ID number, which must be unique for each astronaut. In the main method, line 14 prints out the company name using the variable from the Astronaut class. Remember, a static variable is accessible to all methods in the same class without first creating an object. Notice the instance data fields referenced in the main method use the dot notation to access the data that is not defined as static.

Notice that this class assigns the next ID number to each astronaut and then increases the value by 1.

Quick Check 10-2:

1. Which statement correctly creates a static variable:

- `public static word = "Monday";`
- `public static String word = "Monday";`
- `public String static word = "Monday";`

10.3 Final Static Variables

Often a static variable is also defined as final. This makes sense since there is only one copy of a static variable. Final is another keyword that is used to indicate the variable does not change. The keyword final can be used with or without the keyword static. A non-static final

variable is used for constant values, for example, the definition of `double final PI = 3.14159`; This definition can be used for any constants that need to be defined in our code.

When using the keywords `static final` with a primitive data field, the value cannot be changed. But, it is different is the variable represents an object such as an `ArrayList`. Listing 10.3 shows an example of defining an `ArrayList` as `static final` and then adding values to the `ArrayList`, even though it is defined as `final`. Here is the output that is produced from executing the code in Listing 10.3:

```
[Earth, Mars, Uranus, Venus, Mercury, Jupiter, Saturn, Neptune]
```

Figure 10.2 Sample out from an `ArrayList` defined as `static final`

Listing 10.4 Code used to create an empty `static final ArrayList`, the call `addPlanets` method to update

```
1. import java.util.ArrayList;
2. public class Planets {
3.     private static final ArrayList<String> planets = new ArrayList<String>();
4.     // #A
5.     public static void addPlanets() {
6.         planets.add("Earth"); // #B
7.         planets.add("Mars"); // #B
8.         planets.add("Uranus"); // #B
9.         planets.add("Venus"); // #B
10.        planets.add("Mercury"); // #B
11.        planets.add("Jupiter"); // #B
12.        planets.add("Saturn"); // #B
13.        planets.add("Neptune"); // #B
14.    }
15.    public static void main(String[] args) {
16.        addPlanets(); // #C
17.        System.out.println(planets); // #D
18.        ArrayList dwarfPlanets = new ArrayList<String>();
19.        dwarfPlanets.add("Pluto");
20.        planets = dwarfPlanets;
21.    }
```

#A Create an empty `ArrayList` and define it as `static final`

#B Add planet names to the `ArrayList` even though it was declared as `final`

#C From the main method, call the static method to add the planets to the `ArrayList`

#D Print the `ArrayList` after all planets have been added

In this example, the program can add the values to the `planets ArrayList` even though it has the keywords `static final`. That values can be added because that action does not attempt to update the reference value of the original `planets ArrayList`. In line 19, the code is trying to assign the `planets ArrayList` to the new `ArrayList` of `dwarf planets`, but this attempts to change the reference value and the compiler produces an error "cannot assign a value to final variable `planets`".

It is important to understand the difference between updating the reference value of an object, which is used as the pointer to the location in memory, and the actual values contained in each object such as the list of planets in the example code.

A static variable that includes the attribute `final`, must be initialized before use. This is different from other static variables which are automatically assigned default values. Table 10.2 shows the values that are automatically assigned to a static variable if it not otherwise initialized.

Table 10.2 shows the values assigned to each type of static variable.

Data type	Default value
int, double, float, long, short, byte	0
boolean	false
char	'\u0000' (NUL)
objects	Null

Quick Check 10-3:

1. What is wrong with this code snippet:

```
public class StaticExample {
    private static final String word;
    public static void main(String[] args) {
        System.out.println(word);
    }
}
```

- a. the main method cannot reference the variable `word`
- b. the variable `word` must be initialized
- c. the variable `word` must be declared inside the main method

10.4 Summary

In this lesson, you learned:

- Differentiate between static methods and instance methods
- Understand when to use the `static` keyword
- Correctly define static methods and static variables

This lesson reviewed the difference between static and instance methods, and static and instance variables. This lesson also addressed what happens when we add the keyword `final` to a static variable. In the next lesson, I will introduce the topic of interfaces in Java.

Try this:

Create a Planet class that has the following:

Include the following instance variables: planetName, milesFromEarth, and description

Feel free to use this website to get planet information:
<https://solarsystem.nasa.gov/planets/overview/>

Create a method that converts the distance from earth from miles to kilometers (to keep it simple, use this conversion: 1 M = 1.6 KM)

In the main method, create a new planet object, update the name, distance from earth and the description.

Print the information and include the distance in KM

Here is some sample output:

```
Mars is 138156317.00 miles from Earth and
221050107.20 Kilometers from Earth
It is known as The Red Planet
```

Quick Check 10-1 Solution:

1. Identify which scenarios should be static methods for a Car class:
 - a. getter methods
 - b. setter methods
 - c. accelerate method
 - d. **calculate fuel efficiency method**

Quick Check 10-2:

1. Which statement correctly creates a static variable:
 - a. public static word = "Monday";
 - b. **public static String word = "Monday";**
 - c. public String static word = "Monday";

Quick Check 10-3 Solution:

1. What is wrong with this code snippet:

```
public class StaticExample {
    private static final String word;
    public static void main(String[] args) {
        System.out.println(word);
    }
}
```

- a. **the main method cannot reference the variable word**

- b. the variable word must be initialized
- c. the variable word must be declared inside the main method

Solution to the Try This activity:

10.4 Code for creating a Planet Class

```

1.  public class Planet {
2.
3.      public String planetName;  //#A
4.      public double milesFromEarth;  //#A
5.      public String description;  //#A
6.      public static double convertToKM(double miles) {  //#B
7.          return miles * 1.6;  //#B
8.      }  //#B
9.      public static void main(String[] args) {
10.         Planet mars = new Planet();
11.         mars.milesFromEarth = 138156317;  //#C
12.         mars.description = "The Red Planet";  //#C
13.         mars.planetName = "Mars";  //#C
14.         System.out.printf("%s is %.2f miles from Earth and \n%.2f Kilometers from
    //#D
15.             Earth\n" + "It is known as %s\n\n ", mars.planetName,  //#D
16.                 mars.milesFromEarth, convertToKM(mars.milesFromEarth),
    mars.description);
17.     }  //#D
18. }

```

#A Creates the instance data for a planet

#B Static method used to convert from miles to KM

#C Assign values to the instance data

#D Use print formatting to print out the information about the planet

11

Using Interfaces

After reading lesson 11, you will be able to:

- Understand the definition of an interface
- Create new interfaces
- Implement an interface

This lesson introduces the topic of interfaces in Java. An interface is considered a reference type in Java and serves several purposes. It acts as a rule or contract for any code that implements the interface. All functionality identified in the interface must be implemented. The interface contains a collection of abstract methods, constants, default methods, and static methods. Interfaces are used for dependency injection, unit testing and play a key role in design patterns.

Consider This

Nowadays almost all vehicles contain a navigation system. And, I'm sure you have heard about the latest technology for self-driving cars. It is obvious that the navigation system and the car audio system must communicate with each other.

If we break this process down into the system used by the car manufacturer, which includes the software to provide steering, braking, etc. and another company that provides navigational support, these two companies need some common communication to work together. But the car manufacturer does not need to know the details of the programming behind the task of identifying the best route to the airport and the navigation company does not need to know how the car steers or brakes. They just need to know how to interface with each other, so the car can turn when the navigation says to turn.

So, the programmers devise an interface that acts as a contract between the two software applications. The interface must detail what methods can be invoked to make the car travel. Then the navigation company can write their code to invoke the methods described in the interface to auto drive the car.

11.1 What is an Interface

In Java, an interface is defined similar to a class but it uses the keyword `interface` instead of `class`. It is used to establish a type of contract that describes how the software must interact with any classes that implement this interface. An interface is automatically identified as abstract by the compiler because it does not provide definitions for the methods declared in the interface. In addition, each method must be defined by the implementing class. Each method included in the interface is considered abstract by default. An abstract method is a method that is declared but does not include an implementation. When an abstract method is included in an interface, it only provides the method signature.

Let's start with an example, if we created a video game and we wanted to allow the game to be played on multiple gaming systems, we could create an interface that identifies the specific methods required for the controller to play the game. The gaming systems might include Sony PlayStation, Microsoft Xbox, Nintendo Switch, or even the PC. Each of these manufacturers has a specific type of controller interface that needs to perform the actions identified in the interface, but they each use different buttons, keys, toggles, and so on. Diagram 11.1 shows how the `ControllerInterface` is implemented by each type of controller.

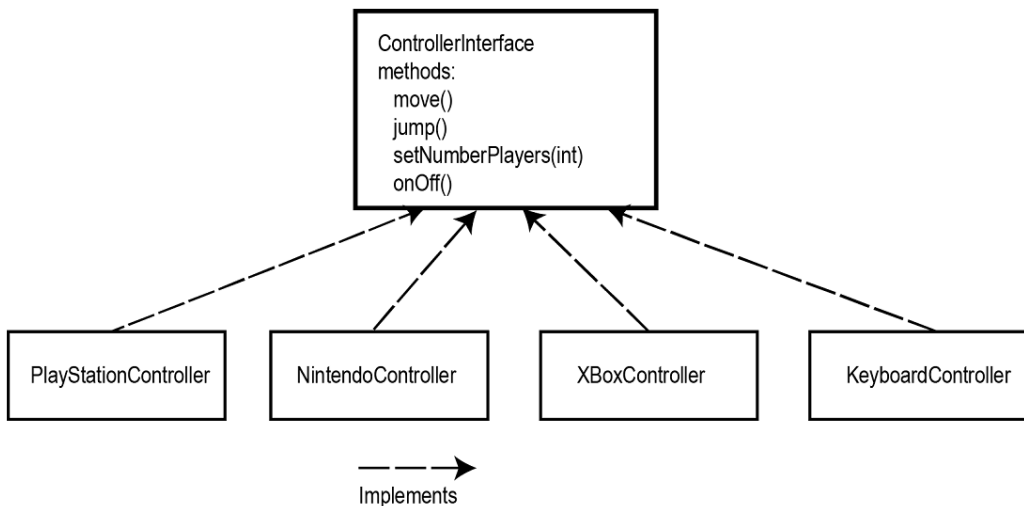


Figure 11.1 shows how each type of controller implements the `ControllerInterface`

Listing 11.1 Code for the ControllerInterface

```

1. public interface ControllerInterface { //A
2.     void onOff(); //B
3.     void move(); //B
4.     void jump(); //B
5.     void setNumberPlayers(int numPlayers); //B
  
```

```
6.     }
```

#A Define an interface

#B Define abstract methods, notice the methods are only signatures, no body is defined here

Listing 11.1 shows the code for the `ControllerInterface`. Notice that the methods only contain the signature, they do not have any code in the body, and they end with a semicolon.

In this example, the controller classes must define how the controller hardware is used. For example, most PC games use the keys W, A, S, and D to control the movement of the characters. But if the user is using an Xbox Controller, then they use the analog stick on the controller. So, the controller classes must implement the logic for each action defined by the interface.

An interface contains only constants, abstract method signatures, default methods, and static methods. Default and static methods contain a method body, but the abstract method signature is just a placeholder for a method that must be defined by the program that is implementing the interface. Java does not require the keywords `public` and `abstract` for the methods, it assigns these values by default. For clarity in this text, I will include both `public` and `abstract`, but remember they are optional. It is also important to note that interfaces cannot be instantiated, which means you cannot create an object with the keyword `new`.

The default method was added with Java 8 to allow programmers to update an interface while still maintaining backwards compatibility with programs that already implement the older version of the interface. The default method must have an implementation defined in the interface.

A benefit of using an interface is that you can implement more than one interface. Another method for declaring classes that share methods is *inheritance*. Java uses inheritance to allow one class to inherit the behaviors of another class. Here is a typical example used to describe this process. If we start with a class representing all animals, this is a very broad category. We don't usually see a rabbit and say "There goes an animal across the grass!". Instead we say, "There goes a rabbit (or bunny) across the grass!". Other examples include dogs, cats, racoons, and so on. The definition of rabbit, dog, cat or racoon are subclasses of the `Animal` class because they share some of the same behaviors and characteristics. So, we create the subclasses for rabbit, dog, etc by extending the animal class. This is referred to as inheritance, the rabbit class inherits from the animal class. A downfall to inheritance is that only one class can be inherited by another class. On the other hand, a class can implement one or many interfaces.

Quick Check 11-1:

1. Which statement correctly uses the `ControllerInterface`:

a. `public class PlayStationController implements ControllerInterface`

- b. `public class PlayStationController extends ControllerInterface`
- c. `public interface PlayStationController implements ControllerInterface`
- d. `public interface PlayStationController extends ControllerInterface`

2. **True/False: An interface is always considered abstract.**

11.2 Defining a new Interface

Defining a new interface is similar to defining a new class. The interface contains a list of methods that need to be defined by each class that implements the interface. It can also include constants, default methods, and static methods. For example, a vehicle interface might need methods to move the car, brake, speed up, and slow down. Let's start by defining the interface using `public interface` followed by the name. Listing 11.1 provides a simple interface definition for a vehicle interface:

11.2 Interface definition for a Vehicle

```

1. public interface VehicleInterface {    // #A
2.     void move(int direction);        // #B
3.     void brake();                    // #B
4.     void accelerate(int speed);      // #B
5.     void slowDown(int speed);       // #B
6. }
```

#A Define an interface

#B Define abstract methods, notice the methods are only signatures, no body is defined here

Remember, each method included in the interface is considered public and abstract by default. Notice that the methods only contain the signature, they do not have any code in the body, and they end with a semicolon. Now, any program that implements our vehicle interface must provide valid definitions for each abstract method. The class that implements the interface must maintain the same return type and parameters (including data type and the number of parameters) as the method signature from the interface.

Quick Check 11-2:

1. Which statement is NOT true about interfaces:
 - a. an interface is always abstract
 - b. an interface can have both instance methods and abstract methods
 - c. a class must use the keyword `implement` to use an interface
 - d. all variables defined in an interface are automatically public, static and final
2. What is wrong with this code snippet:

```

public interface exampleInterface {
    private static final String word;
    public void method1();
}
```

```

public int method2(int num){
    System.out.println(num);
}
}

```

- a. The interface cannot have a static final variable
- b. In the interface, method1 requires a parameter list
- c. Method2 cannot have a method body in the interface, it must be abstract

11.3 Implementing an Interface

Now that we have created an interface, the next step is to write a class that implements this interface. Listing 11.2 is an example of a program that implements the vehicle interface. Figure 11.1 is sample output from executing the code in Listing 11.2.

```

Speed up by 10 mph
Go Straight
Slow down by 5 mph
Turn Right
STOP!!

```

Figure 11.2 Screen print of sample output from executing the code in listing 11.2.

11.3 Implement the VehicleInterface

```

1. public class Vehicle implements VehicleInterface{ // #A
2.     public void move(int direction){ // #B
3.         if(direction == 0)
4.             System.out.println("Turn Right");
5.         else if (direction == 180)
6.             System.out.println("Turn Left");
7.         else if(direction == 90)
8.             System.out.println("Go Straight");
9.         else
10.            System.out.println("Reverse!");
11.    }
12.    public void brake(){ // #B
13.        System.out.println("STOP!!");
14.    }
15.    public void accelerate(int speed){ // #B
16.        System.out.println("Speed up by "+speed + " mph");
17.    }
18.    public void slowDown(int speed){ // #B
19.        System.out.println("Slow down by " + speed + " mph");
20.    }

```

```

21.     public static void main(String[] args) {
22.         VehicleInterface v1 = new Vehicle();           // #C
23.         v1.accelerate(10);                             // #D
24.         v1.move(90);                                   // #D
25.         v1.slowDown(5);                               // #D
26.         v1.move(0);                                   // #D
27.         v1.brake();                                   // #D
28.     }
29. }

```

#A This class implements the VehicleInterface

#B Each abstract method must be implemented

#C Create a new vehicle object. *

#D Invoke each method providing the required arguments

NOTE: Since an interface is considered a reference type, we can assign a variable to the interface, but we must instantiate it with a concrete (not abstract) class.

In addition to implementing interfaces that we define, there are interfaces in the Java API that are very useful to know and understand. When working with objects, it can be extremely helpful to implement the comparable interface. The comparable interface has an abstract method, `compareTo(Object obj)` which returns an int value. The return value is either a negative number, zero, or a positive number as defined here:

If the object in the parameter list is greater than the calling object

return a negative value

else if the object in the parameter list is equal to the calling object

return zero

else return a positive number

For example, if we used Comparable for these two strings:

```
"apples".compareTo("oranges")
```

it would return a negative value since apples comes before oranges alphabetically.

The interface can be used to compare Strings, wrapper class objects and user-defined class objects. This is great when you have a class and you want to include the ability to compare two objects from this class.

When using the comparable interface, the `compareTo(Object obj)` must be defined since it is an abstract method in the interface. To make this method more generic, the compiler understands the keyword `Object` as a datatype. Depending on what part of the object is being compare, it might be necessary to cast the object as a concrete class object. For example, if

we want to compare two vehicle objects, the `compareTo` method defined in the `Vehicle` class can cast the `Object obj` as a `Vehicle` object before starting the comparison:

```
Vehicle v = (Vehicle) obj;
```

In this example, the object in the parameter list is now a `Vehicle` object and can be used to compare to another `Vehicle` object. Code listing 11.3 shows how to use the `Comparable` interface for the `Vehicle` class. Note that the code is implementing two interfaces separated by commas. Multiple interfaces can be implemented, but don't forget to define all abstract methods for each interface.

11.4 Implementing multiple interfaces

```

1.  public class Vehicle implements VehicleInterface, Comparable{   //#A
2.      public String make;
3.      public String model;
4.      public int year;
5.      public String getMake() { return make; }
6.      public void setMake(String make) { this.make = make; }
7.      public String getModel() { return model; }
8.      public void setModel(String model) { this.model = model; }
9.      public int getYear() {return year;}
10.     public void setYear(int year) {this.year = year;}
11.     public void move(int direction){
12.         if(direction == 0)
13.             System.out.println("Turn Right");
14.         else if (direction == 180)
15.             System.out.println("Turn Left");
16.         else if(direction == 90)
17.             System.out.println("Go Straight");
18.         else
19.             System.out.println("Reverse!");
20.     }
21.     public void brake(){
22.         System.out.println("STOP!!");
23.     }
24.     public void accelerate(int speed){
25.         System.out.println("Speed up by "+speed + " mph");
26.     }
27.     public void slowDown(int speed){
28.         System.out.println("Slow down by " + speed + " mph");
29.     }
30.     public int compareTo(Object o) {                               //#B
31.         Vehicle v = (Vehicle)o;                                   //#B
32.         if(this.make == v.make && this.model == v.model &&        //#B
33.            this.year == v.year)                                   //#B
34.             return 0;                                           //#B
35.         else                                                     //#B
36.             return 1;                                           //#B
37.     }
38.     public static void main(String[] args) {
39.         VehicleInterface v1 = new Vehicle();
40.         v1.setMake("Chevy");
41.         v1.setModel("Cruze");
42.         v1.setYear(2018);

```

```

43.     VehicleInterface v2 = new Vehicle();
44.     v1.setMake("Chevy");
45.         v1.setModel("Cruze");
46.     v1.setYear(2017);
47.
48.         if(v1.compareTo(v2) == 0)                ///<#C
49.             System.out.println("These vehicles are the same");    ///<#C
50.         else                                     ///<#C
51.             System.out.println("These vehicles are different");    ///<#C
52.     }
53. }
54.

```

#A This class implements the VehicleInterface AND Comparable

#B Implement the compareTo(Object obj) method for the Vehicle class, this is required

#C Use the new compareTo method to compare two vehicle objects to see if they are the same make, model and year

More interfaces can be found in the Java API, the Comparable interface is often for sorting objects.

Quick Check 11-3:

1. True/False: A class can implement only one interface.
2. Given this interface, which method correctly implements the required abstract methods:

```

public interface SampleInterface {
    //constants
    public int method1();
    public void method2(String s);
}

```

- a. `public int method1(int x) { return x + 5; }`
`public void method2(String s) { return "Welcome " + s; }`
- b. `public int method1() { return 25; }`
`public void method2(String s) { System.out.println("Welcome " + s); }`
- c. `public int method1() { return 25.5; }`
`public void method2(String s) { return true; }`
- d. `public int method1(int x) { return "Your number is " + 5; }`
`public void method2(String s) { return false; }`

11.4 Summary

In this lesson, you learned:

- The definition of an interface
- How to create new interfaces
- To implement an interface

This lesson reviewed how to create and implement interfaces in Java. An interface is a reference type that provides a contract with the program that is implementing it. All abstract

methods from the interface must be defined. Multiple interfaces can be implemented by separating them with commas. In the next lesson, we will work on a cumulative project that includes all of the lessons so far. This is considered a capstone lesson.

Try this:

Use the ControllerInterface from section 11.1 and create the following:

A class that implements the ControllerInterface for each type of controller in figure 11.1

In each class implement the abstract methods from the ControllerInterface

Create a main method that uses the interface as the data type and then instantiate a new object using one of the controller classes

Hint: if you left click on the light bulb icon on line 3, you can choose “**implement all abstract methods**” and NetBeans automatically creates code that you can fill in for each method.

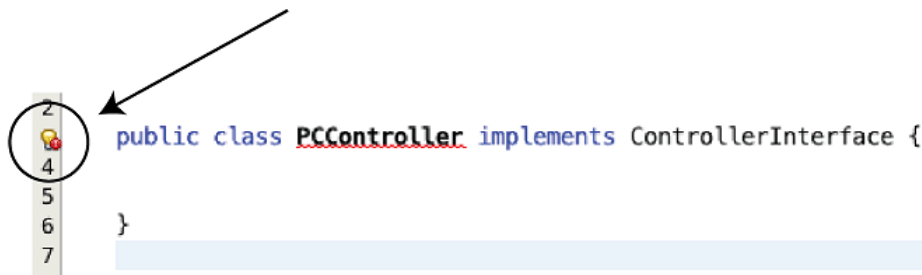


Figure 11.3 Hint for using NetBeans to create code automatically

Quick Check 11-1 Solution:

1. Which statement correctly uses the ControllerInterface:
 - a. `public class PlayStationController implements ControllerInterface`
 - b. `public class PlayStationController extends ControllerInterface`
 - c. `public interface PlayStationController implements ControllerInterface`
 - d. `public interface PlayStationController extends ControllerInterface`
2. True/False: An interface is always considered abstract. **TRUE**

Quick Check 11-2:

1. What is wrong with this code snippet:

```
public interface exampleInterface {
    private static final String word;
```



```

public void method1();
public int method2(int num){
    System.out.println(num);
}
}

```

- The interface cannot have a static final variable
- In the interface, method1 requires a parameter list
- Method2 cannot be defined in the interface, it must be abstract**

Quick Check 11-3 Solution:

- True/False: A class can implement only one interface. **FALSE**
- Given this interface, which method correctly implements the required abstract methods:

```

public interface SampleInterface {
    //constants
    public int method1();
    public void method2(String s);
}

```

- ```

public int method1(int x) { return x + 5; }
public void method2(String s) { return "Welcome " + s; }

```
- ```

public int method1() { return 25; }
public void method2(String s) { System.out.println("Welcome " + s); }

```
- ```

public int method1() { return 25.5; }
public void method2(String s) { return true; }

```
- ```

public int method1(int x) { return "Your number is " + 5; }
public void method2(String s) { return false; }

```

Solution to the Try This activity:

Note: for this solution, I have placed each class and interface in their own files. They are connected by the package name: `package videogames;`

Listing 11.5 Solution to Try This Activity

```

1. package videogames; //A
2.
3. public class VideoGames {
4.     public static void main(String[] args) {
5.         ControllerInterface pc = new PCController(); //B
6.         pc.onOff(); //C
7.         ControllerInterface xbox = new XboxController(); //B
8.         xbox.onOff();//C
9.         ControllerInterface ps = new PlayStationController(); //B
10.        ps.onOff();//C
11.        ControllerInterface nintendo = new NintendoController();//B
12.        nintendo.onOff();//C
13.    }
14. }

```

```

1. package videogames;
2.
3. public class PCController implements ControllerInterface {
4.     public void onOff() { System.out.println("PC Keyboard turn on/off");}
5.     public void move() { System.out.println("Move");}
6.     public void jump() { System.out.println("Jump"); }
7.     public void setNumberPlayers(int numPlayers) {this.numPlayers =
numPlayers;}
8.     private int numPlayers = 0;
9. }

1. package videogames;
2.
3. public class XboxController implements ControllerInterface{
4.     public void onOff() { System.out.println("XBox turn on/off");}
5.     public void move() { System.out.println("Move");}
6.     public void jump() { System.out.println("Jump"); }
7.     public void setNumberPlayers(int numPlayers) {this.numPlayers =
numPlayers;}
8.     private int numPlayers = 0;
9. }

1. package videogames;
2.
3. public class NintendoController implements ControllerInterface{
4.     public void onOff() { System.out.println("Nintendo turn on/off");}
5.     public void move() { System.out.println("Move");}
6.     public void jump() { System.out.println("Jump"); }
7.     public void setNumberPlayers(int numPlayers) {this.numPlayers =
numPlayers;}
8.     private int numPlayers = 0;
9. }

1. package videogames;
2.
3. public class PlayStationController implements ControllerInterface{
4.     public void onOff() { System.out.println("PlayStation turn on/off");}
5.     public void move() { System.out.println("Move");}
6.     public void jump() { System.out.println("Jump"); }
7.     public void setNumberPlayers(int numPlayers) {this.numPlayers =
numPlayers;}
8.     private int numPlayers = 0;
9. }

```

#A This is the package name that groups all the files together in one application

#B Declare an object for each type of controller

#C Invoke the on/off method to print a message for each controller

Each class has its own file: PCController.java, XboxController.java, VideoGames.java, NintendoController.java, and PlayStationController.java They are all connected using the same package name: videogames.

12

Capstone

In lessons 8 - 11, I discussed how to access and use the Standard Java API, the difference between the String and StringBuilder class, static methods and variables, and finally the definition and uses of interfaces in Java.

This lesson is a capstone of these topics providing you with the opportunity to reinforce the skills learned. In this lesson, I will introduce a problem that is designed to include everything presented so far in the book.

This assignment involves creating an application for a book publishing company, ACME Publishing. ACME publishing offers both hardcopy and e-books. Here are the requirements for this assignment:

- Obtain the author information for the book including:
 - author id, author name, address, email address, and phone number
- Allow the application to create book objects with the following information:
 - author id, book title, number of pages
- When the user enters the book information, it can choose either 'h' for hard copy, 'e' for electronic copy, or 'b' for both
- Allow the user to enter the author and book information from the console as one line of String of data for each
- Print out the author and book information using a StringBuilder object

As I've stated before, everyone programs differently. So, the order in which you create the classes for this application might vary from my approach. I have decided to start by creating an Author class, then a Book class. Finally, I'll add an ACMEPublishing class which contains the main method. For this project, there is no need for an interface.

12.1 Create the Author class

For this example, ask the user to enter the information for each author. The instance data for this class includes:

- Author's first and last name
- Address
- Email address
- Phone number
- Author ID (each other is assigned a unique author id starting at 001)

The user must be prompted to enter the author information on one line separated by commas. For example: Peggy,Fisher,123 Main Street,pegfisher@aol.com,717-555-1212.

The Author class must also assign a unique Author ID to each author. To make sure we only have one place where the **next** author id is stored, the Author class must create a static int variable, ie: `static int nextAuthorID = 001;` which is increased by 1 after it is assigned to an author. In addition, since every book must be associated with a unique author, we need a get method to retrieve the author id so that it can be used when creating a book object (covered in section 12.3). It is a good idea to supply get and set methods for all instance data, but to save space, I have only added the get method for the `authorID` field.

The last method that is included in the Author class is a `toString` method that is used to format the author information. By default, all classes have access to a `toString` method, it is included in the Object class. But the default `toString` method simply prints the reference value of the object. Instead, it is often helpful to add a specific method that is used in place of the default method. The use of the `@Override` keyword allows the class to provide a substitute implementation of the `toString` method. Figure 12.1 shows sample output of how the Author class is formatting the author information.

```
Enter author first name, last name, address, email, and phone number
Peggy,Fisher,123 Main Street,pegfisher@aol.com,717-555-1212
```

```
Author Information
=====
Peggy Fisher
123 Main Street
email: pegfisher@aol.com
Author ID: 1
```

Figure 12.1 Sample output for printing the author information

Listing 12.1 shows my implementation of the Author class

Listing 12.1 Author Class

```
1. package acmepublishing; // #A
2. public class Author {
```

```

3.     public static int nextAuthorID = 001;    //#B
4.     private String lName, fName, address, email, phone;    //#C
5.     private int authorID;    //#C
6.     public Author(String fName, String lName, String address,    //#D
7.         String email, String phone) {    //#D
8.         this.lName = lName;    //#D
9.         this.fName = fName;    //#D
10.        this.address = address;    //#D
11.        this.email = email;    //#D
12.        this.phone = phone;    //#D
13.        this.authorID = nextAuthorID++;    //#D
14.    }    //#D
15.    public int getID() {    //#E
16.        return authorID;
17.    }
18.    //add other getter and setter methods here
19.    @Override    //#F
20.    public String toString() {    //#F
21.        return "\n\nAuthor Information\n=====\n" + fName + " " +
22.            lName + "\n" + address + "\nemail: "+email + "\nAuthor ID: " +
23.            authorID;    //#F
24.    }    //#F
25. }

```

#A All classes have the same package name, this allows each class to be in a separate file

#B Create a static variable for the next author id, this variable is shared with all author objects

#C Create instance variables for the author information

#D Create a method used to populate the instance data for a specific author, this special method is called a constructor and more details are included in Unit 3

#E The `getter` method for `authorID` field

#F Override the `toString` method to format the author information when printed

NOTE: I feel it is important to note that this application does not include extensive error checking for any fields that could have invalid data. We covered error checking for instance data fields in Lesson 4.2. Please refer to that lesson to get an idea of how we added code to validate the values entered by the user to enforce data integrity. Error checking should be added to all instance data fields as appropriate, for example, the phone number field should be check for 10 digits (with or without hyphens and parentheses). Due to the additional length of the code if all of the error checking was added, I decided to keep the programs simple. But a production version of these programs must include error checking.

12.2 Create the Book class

Next, ask the user to enter the information for each book this author has created. The instance data for this class includes:

- Book title
- Book type ('h' – hard copy, 'e' – electronic copy, 'b' – both)
- Number of pages

The Book class also needs the `authorID`, but this value is supplied using the ID created when the Author object was created. The user must be prompted to enter the book information on

one line separated by commas. For example: Get Programming with Java,b,325. This represents the title "Get Programming with Java" which is published as hard copy and electronic, and has 325 pages.

When the main method creates a new book object, it uses the data entered by the user to pass values in the argument list that are used to populate the instance data. It also uses the `getAuthorID` method from the Author class so it can link the author to this book. Similar to the abbreviated list of getter and setter methods in the Author class, I've decided to omit the list of getter and setter methods for the Book class to save room in the text. The only get method is defined to get the author id to print all books associated with that author.

For the Book class, I have also provided an override method for the `toString` method. Figure 12.2 shows sample output of how the Book class is formatted with the book information. Listing 12.2 shows a sample Book Class.

```
Book Info:
Author ID: 1
Book Title: Get Programming with Java
Book Type: b
Number of Pages: 325
```

Figure 12.2 Sample output for print the book information

Listing 12.2 Book Class

```
1. package acmepublishing;    //#A
2. public class Book {
3.     private int authorID;    //#B
4.     private String bookTitle;    //#B
5.     private char bookType = 'h'; //h-hardcopy, e-ebook    //#B
6.     private int numPages;    //#B
7.     public Book(int authorID, String bookTitle, char bookType, int numPages) {
8.         this.authorID = authorID;
9.         this.bookTitle = bookTitle;
10.        if(bookType == 'e' || bookType == 'h' || bookType == 'b')    //#C
11.            this.bookType = bookType;    //#C
12.        else
13.            this.bookType = 'h'; //if an invalid book type is entered,    //#C
14.                               //it is set to hard copy    //#C
15.        this.numPages = numPages;
16.    }
17.    public int getAuthorID() {return authorID;}
18.    //Getter and Setter methods go here
19.    @Override
20.    public String toString() {    //#D
21.        String type;
22.        if(bookType == 'e')
23.            type = "Electronic copy only";
```

```

24.         else if(bookType == 'h')
25.             type = "Hard copy only";
26.         else
27.             type = "Both hard copy and electronic";
28.         return "\n\nBook Info: \nAuthor ID: " + authorID + "\nBook Title: "+
29.             bookTitle + "\nBook Type: " + bookType + "\nNumber of Pages: "
30.             + numPages;
31.     }    // #D
32. }

```

#A All classes have the same package name, this allows each class to be in a separate file

#B Create instance variables for the book information

#C Sample error logic has been added to test for a correct book type of 'h', 'e', or 'b'

#D Override the toString method to format the book information when printed

12.3 Create the ACMEPublishing class

Now, we are ready to define the ACMEPublishing class which includes the main method. Remember, a Java application can only have one main method. This is the starting point for the application. This method is used to create Author objects along with one or more book objects for each author. The Author objects are added to one ArrayList and the book objects are added to another. Since each book object contains the author id, we can match the books with the author when printing.

The main method includes the following tasks:

- Create both ArrayLists
- Create a loop to allow the user to enter one or more authors
- Prompt the user to enter the author information
- Read all information into one String object
- Using the String split method, split the information into a String array
- Create a new author object and provide the author data as arguments
- Add the new author object to the ArrayList
- Next, follow the same process for the author's book
- Allow the user to enter in multiple books for each author, assigning the same author id to each book
- Finally, use a StringBuilder object to create a String containing all the authors and the books associated with each other
- Print out the result

There is one major difference when working with the Strings for the book object. Since the Book class includes an `int` variable for the number of pages, and a `char` variable for the book type, these values must be converted from String to the appropriate data type. For the `int` value, I am using the Integer wrapper class and the `parseInt` method. For the character, I am using the `charAt` method for a String object to retrieve the character.

It is always good to think about the types of data that the user might input. For each book object, the user needs to enter the title, type, and number of pages. In order to correctly convert the type and number of pages to their respective values, it is a good idea to strip out any extra white space (which includes spaces). For my version of the main method, I am using the `replaceAll` method from the `String` class: `replaceAll(" ", "")`.

HINT: Remember, strings are immutable, so when you use the `replaceAll`, you must create a new string to hold the new value.

Figure 12.3 shows the sample output from running the `ACMEPublishing` application and Listing 12.3 shows the sample code used to produce this output.


```

Enter author first name, last name, address, email, and phone number
Peg,Fisher,123 Main Street,pegfisher@aol.com,717-555-1212
Enter book information: Book Title, Book Type (h-hardcopy, e-ebook) and number of pages
Get Programming with Java,b,325
Does this author have any more books?
y
Enter book information: Book Title, Book Type (h-hardcopy, e-ebook) and number of pages
Get Programming with C++,h,435
Does this author have any more books?
n
Is there another author?
y
Enter author first name, last name, address, email, and phone number
Cy,Fisher,455 Bonnie Lane,cyfisher@embarqmail.com,717-324-9090
Enter book information: Book Title, Book Type (h-hardcopy, e-ebook) and number of pages
Ethical Cyber Hacking, b, 362
Does this author have any more books?
n
Is there another author?
n

```

```

Author Information
=====

```

```

Peg Fisher
123 Main Street
email: pegfisher@aol.com
Author ID: 1

```

```

Book Info:
Author ID: 1
Book Title: Get Programming with Java
Book Type: b
Number of Pages: 325

```

```

Book Info:
Author ID: 1
Book Title: Get Programming with C++
Book Type: h
Number of Pages: 435

```

```

Author Information
=====

```

```

Cy Fisher
455 Bonnie Lane
email: cyfisher@embarqmail.com
Author ID: 2

```

```

Book Info:
Author ID: 2
Book Title: Ethical Cyber Hacking
Book Type: b
Number of Pages: 362

```

Figure 12.3 Sample output from the ACMEPublishing application

Listing 12.3 ACMEPublishing Class

```

1. package acmepublishing;
2. import java.util.Scanner; //A
3. import java.util.ArrayList; //A
4.
5. public class ACMEPublishing {
6.     public static void main(String[] args) { //B
7.         Scanner in = new Scanner(System.in);
8.         boolean moreAuthors = true;
9.         boolean moreBooks = true;
10.        String response;
11.        ArrayList<Author> authors = new ArrayList<Author>(); //C
12.        ArrayList<Book> books = new ArrayList<Book>();//C
13.        while (moreAuthors) {
14.            System.out.println("Enter author first name, last name, "
15.                + "address, email, and phone number");
16.            String authorData = in.nextLine(); //D
17.            String[] authorInfo = authorData.split(",");//E
18.            Author a = new Author(authorInfo[0], authorInfo[1], authorInfo[2],
19.                authorInfo[3], authorInfo[4]); //F
20.            authors.add(a); //F
21.            while (moreBooks) {
22.                System.out.println("Enter book information: Book Title, "
23.                    + "Book Type (h-hardcopy, e-ebook) "
24.                    + "and number of pages");
25.                String bookData = in.nextLine();
26.                String[] bookInfo = bookData.split(","); //E
27.                String bookType = bookInfo[1].replaceAll(" ", ""); //G
28.                int numPages = Integer.parseInt(bookInfo[2].replaceAll("
", //H
29.                    ""));
30.                Book b = new Book(a.getID(), bookInfo[0], //F
31.                    bookType.charAt(0), numPages);
32.                books.add(b); //F
33.                System.out.println("Does this author have any more books? ");
34.                response = in.next();
35.                in.nextLine();
36.                if (response.charAt(0) != 'Y' && response.charAt(0) != 'y') {
37.                    moreBooks = false;
38.                }
39.            }
40.            System.out.println("Is there another author? ");
41.            response = in.next();
42.            in.nextLine();
43.            if (response.charAt(0) != 'Y' && response.charAt(0) != 'y') {
44.                moreAuthors = false;
45.            }
46.            else
47.                moreBooks = true; //reset for next author
48.        }
49.        StringBuilder printInfo = new StringBuilder(); //I
50.        for(Author a:authors) {
51.            printInfo.append(a.toString());
52.            for(Book b:books) {
53.                if(b.getAuthorID()==a.getID())//J
54.                    printInfo.append(b.toString());

```

```

55.         }
56.     }
57.     System.out.println(printInfo);
58. }
59. }

```

```

#A Import the Scanner, and ArrayList classes from the Standard Java API
#B Start of the main method
#C Create the two ArrayList objects to hold the list of authors and books
#D Read in the entire line of data separated by commas
#E Store each separate data field in a string array using the comma as the delimiter
#F Add the object to the ArrayList
#G Remove spaces from the book type data fields
#H Convert the string for the number of pages to an integer after removing any spaces
#I Create a StringBuilder object to hold the output
#J After adding the author to the StringBuilder object, loop through all the books to find any books with the same
    author id

```

12.4 Summary

This lesson provided an activity that included all the topics from lessons 8-11. In this lesson, I reviewed the initial business problem of creating a book publishing application that asked the user to enter the author information followed by one or more books they have written.

The requirements for this project included:

- Obtainin`g and parsing the author information for the book including:
 - author id, author name, address, email address, and phone number
- Allowing the application to create book objects with the following information:
 - author id, book title, book type, number of pages
- Printing out the author and the book information

In the next unit, I will continue to work with classes and objects by presenting a real-world example, a banking application.

Try this:

Create a Student class that has the following:

- Include the following instance variables: studentName, GPA, collegeName (where the college is always "Penn State University")
- Implement the Comparable Interface
- Find the average GPA for all students
- Read information about each student from the console, read it as strings separated by commas (first name, last name, and GPA)
- Assign each student a unique student ID
- In the main method, create student objects, store these objects in an ArrayList
- Print a roster of all students and student id

- Here is some sample output:

```

Enter the student information (first name, last name,GPA
Joe, Fusco, 3.5
More students: (Y/N)
y
Enter the student information (first name, last name,GPA
Cy, Fisher, 4.0
More students: (Y/N)
y
Enter the student information (first name, last name,GPA
Owen,Damincantonio,3.95
More students: (Y/N)
y
Enter the student information (first name, last name,GPA
Riley,Cullen,3.25
More students: (Y/N)
y
Enter the student information (first name, last name,GPA
Stephanie,Evans,3.0
More students: (Y/N)
n
Joe Fusco, Student ID: 505 GPA: 3.5, University: Penn State University
Cy Fisher, Student ID: 505 GPA: 4.0, University: Penn State University
Owen Damincantonio, Student ID: 505 GPA: 3.95, University: Penn State University
Riley Cullen, Student ID: 505 GPA: 3.25, University: Penn State University
Stephanie Evans, Student ID: 505 GPA: 3.0, University: Penn State University

The student with the highest GPA is: Cy Fisher, Student ID: 505 GPA: 4.0, University: Penn State University

Average GPA is: 3.54

```

Figure 12.4 Sample output from executing the Student class program

Solution to the Try This activity:

12.6 Code for creating a Student class

```

1. import java.util.ArrayList;
2. import java.util.Scanner;
3. public class Student implements Comparable{ //A
4.     static Scanner in = new Scanner(System.in);
5.     static int ID = 500; //B
6.     private String fName, lName; //C
7.     private int studentId; //C
8.     private double GPA; //C
9.     static String collegeName = "Penn State University"; //D
10.
11.     public void readStudentInfo() { //E
12.         System.out.println("Enter the student information (first name, last
name, "
13.             + "GPA");
14.         String info = in.nextLine(); //E
15.         String[] parsedInfo = info.split(","); //F
16.         this.fName = parsedInfo[0]; //G
17.         this.lName = parsedInfo[1]; //G
18.         this.GPA = Double.parseDouble(parsedInfo[2]); //G
19.         this.studentId = ID++; //H
20.

```

```

21.     }
22.     public int compareTo(Object obj) { //#I
23.         Student s = (Student)obj; //#I
24.         if(this.GPA > s.GPA) //#I
25.             return 1; //#I
26.         else //#I
27.             return -1; //#I
28.     } //#I
29.     @Override
30.     public String toString() {
31.         return fName + " " + lName + ", Student ID: " + ID + " GPA: " + GPA +
32.             ",University: " + collegeName + "\n";
33.     }
34.     public static void main(String[] args) {
35.         boolean done = false;
36.         String response = "y";
37.         ArrayList<Student> roster = new ArrayList<Student>();
38.         while(done == false) {
39.             Student s = new Student();
40.             s.readStudentInfo();
41.             roster.add(s);
42.             System.out.println("More students: (Y/N)");
43.             response = in.next().toLowerCase();
44.             in.nextLine();
45.             if(response.equals("n"))
46.                 done = true;
47.         }
48.         StringBuilder output = new StringBuilder(); //#J
49.         for (Student s_info : roster) {
50.             output.append(s_info.toString()); //#J
51.         }
52.         System.out.println(output);
53.         double highest = roster.get(0).GPA;
54.         int highestIndex = 0;
55.         double total = roster.get(0).GPA;
56.         for(int i = 1; i < roster.size(); i++) {
57.             total += roster.get(i).GPA;
58.             if(roster.get(i).GPA > highest) {
59.                 highestIndex = i;
60.                 highest = roster.get(i).GPA;
61.             }
62.         }
63.         average = total/roster.size();
64.         System.out.println("The student with the highest GPA is: "
65.             + roster.get(highestIndex).toString());
66.         System.out.println("Average GPA is: "+average);
67.     }
68. }

```

#A Implement the Comparable Interface

#B Create a static variable to use for assigning student IDs

#C Create instance data for the student objects

#D Create a static variable for the college name since all students attend Penn State University

#E Read a string of input data from the user for each student

#F Add the data fields to a string array using a comma as the delimiter

#G Update the instance data with the values from the string array

#H Assign the student an ID number, then add one to the static student id field
#I Implement the `compareTo` method to compare the GPA of two students
#J Create a `StringBuilder` object to store all the output that needs to be printed

Unit 3

Programming with Objects

13

Overloading Methods

After reading lesson 13, you will be able to:

- Describe three ways to overload a method
- Write a class that contains an overloaded method(s)
- Call the overloaded method(s)
- Create overloaded constructor methods

This lesson introduces the topic of overloading methods in Java. In Java, a class cannot have two methods with the exact same method signature because the compiler would not be able to distinguish between the two methods. The Java compiler checks for this at compile time and generates an error message when this happens. But there are situations where the programmer needs to perform a similar function that is just slightly different, for example, maybe the number of values that are passed through as arguments to the method vary. To keep the code readable, it is helpful to give these methods the same name but allow them to have different method signatures.

Before discussing method overloading further, let's review the nomenclature for a method signature. A parameter is a variable in a method signature, it includes a data type and variable. When a method is called, the arguments refer to the data you pass into the method's parameters. Figure 13.1 shows a code snippet describing the arguments as the variables passed to the method and the parameters are the variables that receive these values and are made available to the statements inside the method.

in the main method, there is a call to the `calculateAverage` method:

```
int a = 5;
int b = 14;
double avg = calculateAverage(a, b);

public double calculateAverage(int num1, int num2){
    ...
}
```

Figure 13.1 Code snippet identifying the arguments and parameters for a method call

Consider This

The postal service has certain elements that must be included when mailing a letter or postcard, but those elements can differ if the mail is going to a PO Box, an apartment, a house address, a military address, or an international address to name a few.

Many companies outsource their marketing efforts by providing mailing lists that another company uses to print postcards or mail letters to their current and potential customers. The marketing company must have different methods to print the addresses based on the values in the file. For example, the file might contain a first name, last name, address line 1, address line 2, city, state, and zip code (with or without an extended zip code). If the file contains all of these elements, then the method is expecting seven values from the calling program. But not every customer has an address line 2, so a second method signature can have the same values except for a `String` variable for the second address line.

Both methods need to print a mailing label, but the information in the argument list might be different.

13.1 Three ways to overload a method

There are three ways to overload a method (see matching examples in figure 13.2):

1. Varying the number of parameters in the method signature
2. Changing the data type of one or more parameters in the method signature
3. Changing the order of the parameters in the method signature

```
public void printPage(){...} //version 0
v.1 public void printPage(int pageNum, String title) {...}
v.2 public void printPage(double pageNum, String title){...}
v.3 public void printPage(String title, int pageNum){...}
```

Figure 13.2 Examples of the three ways to overload the same method

The example above shows the original method called `printPage` which does not have any arguments and does not return any values. Each subsequent method is slightly different, v.1 shows the same method name but this time with two arguments: `int pageNum` and `String`

title. Next, v.2 changes the data type of the first variable from `int` to `double`, this is enough to overload the method. Finally, v.3 reverses the arguments so that the `String` value comes before the `int` value.

The compiler determines which method to call based on the values passed as arguments to the method. Figure 13.3 shows an example of a statement that invokes each of the methods from figure 13.2.

```
printPage();           → public void printPage(){...}
printPage(10, "Alice in Wonderland"); → public void printPage(int pageNum, String title) {...}
printPage(10.5, "Maze Runner"); → public void printPage(double pageNum, String title){...}
printPage("Eragon", 130); → public void printPage(String title, int pageNum){...}
```

Figure 13.3 matches a method call to the corresponding overloaded method.

Note

the names of the parameter variables are not used to determine if a method is overloaded, only the data type of the parameter, the number of parameters, and the order of the parameters are important when overloading a method. The variable names can be the same or different.

It is also important to note that changing the return type is **not** considered overloading a method. So, if I changed the method signature to: `public String printPage(){...}`, this would cause a compile time error since the compiler would not be able to differentiate the methods.

In other words, if the method signature is exactly the same, but the return type is different, this will cause a compile time error. It is not considered an overloaded method if **ONLY** the return type is different. The problem is that the compiler would not know which method to invoke since it only checks for the signature and it does not know what type of value is returned inside the method.

Figure 13.4 is an example where the compiler produces an error message for the second method signature since the only change is the return type. The error message is:

```
method printPage() is already defined in class Lesson13_Example.
```

```
public void printPage(){...}
public String printPage() {...} //ERROR
public String printPage(int pageNum, String title){...} //OK
```

Figure 13.4 is an example of an error message generated when incorrectly overloading the `printPage` method

In the next section I will show some examples of overloaded methods and the code that calls these methods. Remember, when writing an overloaded method, the compiler checks to make sure the method signature is not already defined. The compiler also checks to make sure that there is a method that matches the arguments in the statement calling the method to the parameters in the method signature, similar to the example in figure 13.3.

Overloading vs. Overriding

Method overloading only occurs within the same class file. It is used to provide the same method name for multiple methods, but each method contains differences in the parameter list. Method overriding occurs when a subclass contains the exact same method signature as a method in the superclass.

Method overloading is sometimes referred to as static polymorphism, which is a fancy way of saying that it is checked at compile time, not run-time. Method overriding is a form of run-time polymorphism. This indicates that the method to be invoked is not determined until run-time.

Although overriding is introduced in detail in the next lesson, I wanted to explain the difference between these two concepts here.

When a variable is created using a superclass, it can be assigned a reference value to an object that is a subclass of that superclass. In this example, the subclass contains a method that overrides the same method in the superclass. So, the call to the overridden method is determined at runtime since the compiler does not necessarily know whether the reference object is to the superclass or to the subclass. This process in which a call to the overridden method is resolved at runtime is known as dynamic binding or run-time polymorphism.

13.2 Quick Check 13-1:

1. Given the method signature, `public int getPageNumbers()`, which method signature does **not** correctly overload this method:
 - a. `public int getPageNumbers(String title)`
 - b. `public void getPageNumbers(int num)`
 - c. `public double getPageNumbers()`
 - d. `public int getPageNumbers(int num)`

2. Given the method signature, `public double getAverage(int num1, int num2, int num3)`, which method signature(s) correctly overload this method:
 - a. `public double getAverage(double num1, int num2, int num3)`
 - b. `public double getAverage(double num1, double num2, double num3)`
 - c. `public double getAverage(int num1, double num2, int num3)`
 - d. All of the above

13.3 Write a class with overloaded methods

In the “Consider This” section of this lesson, I presented the idea of writing an application to print mailing addresses given a file with names and addresses. Now that we understand the different ways to overload a method, let’s create a class that can be used to print several different types of addresses. In this example, I will provide three overloaded methods, each method is name is `print`. Each method contains a different number of parameters reflecting the scenarios listed below:

1. First and last name, address line, city, state, zip code

Peggy Fisher

*123 Main Street
Ambler, PA 19001*

2. Only address line, city, state, zip code (instead of name, use: *Current Resident*)

*Current Resident
123 First Street
Apartment B
Carpinteria, CA 23901*

3. First and last name, PO box, city, state, zip code

*Tish Macclay
PO Box 756
Scituate, MA 02566*

Listing 13.1 shows the start of our class along with the three methods required for the scenarios listed above.

Note:

When creating the `print` methods for the `Address` class, I have chosen to use the data type `String` for the zip code. Although a five-digit zip code is all numeric, if the data type was an `int`, the `print` statement would require extra formatting to print out any leading zeroes. And, many post offices are now requiring the extended zip code which is in a format: 99999-9999, so using a `String` data type makes it easy to adjust if any address records have the extended zip code.

Listing 13.1 Code for overloading three print methods to print an address

```

1  public class Address {
2      public void print(String fName, String lName, String    //#A
3          address1, String city, String state, String zip) {
4          //code to print address omitted
5      }
6      public void print(String address1, String address2, String city, String
7          state,
8          String zip) { // #B
9          //code to print address omitted
10     }
11     public void print(String fName, String lName, int POBox,
12         String city, String state, String zip) {
13         //code to print address omitted
14     }

```

#A This method has 6 `String` arguments

#B This method only has five `String` arguments

#C This method also has 6 arguments but the order is different: two `Strings`, one `int`, and three more `String` values

It is important to look closely at the examples above. Notice that the first and third methods have the exact same return type, method name and number of arguments. The difference is the data types of the argument list. The third method has an integer value for the PO Box and one less `String` value (it does not need address line 1 since it is using a PO Box). That is enough to correctly overload the first method so there are no compile errors in this code. Remember, a method can be overloaded by:

- Containing a different number of parameters
- Having different data types for one or more of the parameters
- Changing the order of the parameters

In the next section, I will show you examples of using each of the three methods to generate a mailing list.

Quick Check 13-2:

1. Given the following class:

```
class Student {
    public void printStudent(int studentNum, String fname, String
        lname, double GPA) {
        //code to print student info goes here
    }
    public void printStudent(int studentNum, String name, String
        studentMajor, double GPA) {
        //code to print student info goes here
    }
}
```

Why won't this code compile?

- a. It does compile, there are no errors
 - b. Both methods cannot have a void return type
 - c. Both methods cannot have the same number of arguments
 - d. The second method does not overload the first since the data types are all the same
2. Using the code from the first problem, which statement correctly provides the start of an overloaded method:
 - a. `public String printStudent(int studentNum, String fname, String lname, double GPA)`
 - b. `public void print(int studentNum, String fname, String lname, double GPA)`
 - c. `public void printStudent()`
 - d. `public String print()`

13.4 Call the overloaded method

Now that we have created the overloaded methods, the next step is to call these methods. This is where the real value of overloaded methods comes into play. When an object is instantiated in a class that has overloaded methods, the correct method is determined by the quantity, data type and order of the argument list.

Let's use the Address class in Listing 13.1 as a starting point for examples on calling the overloaded methods. There are three versions of the print method, each overloaded by changing the parameter list. In the main method, I will create an Address object and then use the print method to print three different types of addresses. Figure 13.5 shows sample output from executing the code in Listing 13.3.

Listing 13.1 Code for overloading three print methods to print an address

```

1     public class Address {
2         public void print(String fName, String lName, String    //#A
3             address1, String city, String state, String zip) {
4             //code to print address omitted
5         }
6         public void print(String address1, String address2, String city, String
7             state,
8                 String zip) { //#B
9             //code to print address omitted
10        }
11        public void print(String fName, String lName, int POBox,
12            String city, String state, String zip) {
13            //code to print address omitted
14        }
15    }

```

#A This method has 6 String arguments

#B This method only has five String arguments

#C This method also has 6 arguments but the order is different: two Strings, one int, and three more String values

```

Current Resident
123 First Street
Carpinteria, CA 23901

```

```

Peggy Fisher
123 Main Street
Ambler, PA 19001

```

```

Tish Macclay
PO Box 756
Scituate, MA 02566

```

Figure 13.5 Sample output printing three addresses using overloaded methods

Listing 13.2 Complete program to print three addresses using overloaded methods

```

1   public class Address {
2       public void print( String fName, String lName, String
3           address1, String city, String state, String zip) {
4           System.out.println(fName + " "+lName+"\n"+address1 +
5               "\n"+city+", "+state+" "+zip+"\n");
6       }
7       public void print(String address1, String city, String state, String zip)
8       {
9           System.out.println("Current Resident\n"+address1 +
10              "\n"+city+", "+state+" "+zip+"\n");
11      }
12      public void print(String fName, String lName, int POBox, String city,
13          String state, String zip) {
14          System.out.println(fName + " "+lName+"\nPO Box "+POBox +
15              "\n"+city+", "+state+" "+zip+"\n");
16      }
17      public static void main(String[] args){
18          Address address1 = new Address();    //A
19          address1.print("123 First Street", "Carpinteria", "CA", "23901"); //B
20          address1.print("Peggy" ,"Fisher", "123 Main Street", "Ambler",
21              //C
22              "PA", "19001");
23          address1.print("Tish","Macclay", 756, "Scituate", "MA", "02566"); //D
24      }
25  }

```

#A Instantiate an Address object

#B Call the overloaded method that uses Current Resident as the name

#C Call the overloaded method that takes six String arguments

#D Call the overloaded method that takes five String arguments and a numeric value for the PO Box

Quick Check 13-3:

For this quick check, use the following code snippet to answer the questions

Listing 13.3 Code for a ToDoList application

```

1   public class ToDoList {
2       String task;
3       int priority;
4       boolean repeat;
5       public ToDoList(String task, int priority, boolean repeat){
6           this.task = task;
7           this.priority = priority;
8           this.repeat = repeat;
9       }
10      public void print(String name){
11          System.out.println(task + ", priority: "+ priority+
12              ", repeat: "+repeat + "\nAssigned to: "+name);
13      }
14      public void print(){
15          System.out.println(task + ", priority: "+ priority+
16              ", repeat: "+repeat);
17      }

```

```

18     public static void main(String[] args){
19         ToDoList todo = new ToDoList("Homework", 1, true);
20         todo.print();
21         todo.print("Bob Fisher");
22     }
23 }

```

1. What is the output if this statement is added to the main method:

```
todo.print("Peggy Fisher");
```

- a. Homework, priority: 1, repeat: true
Assigned to: Peggy Fisher
 - b. No output is produced, this causes an error
 - c. Homework, priority: 1, repeat: true
 - d. Homework, priority: 1, repeat: false
2. Which line(s) of code are considered overloaded methods?
- a. 5, 10, 14
 - b. 10, 14
 - c. 5
 - d. 10
3. What is wrong with adding this statement: `todo.print("Mow the lawn", "Peggy", 5, true);`
- a. Nothing, this is correct
 - b. The constructor does not take a name and task
 - c. There is no overloaded print method for these arguments
 - d. Mow the lawn must be switched with the name Peggy

13.5 Overloading Constructors

Methods that are used to instantiate an object are called constructors. These methods are frequently overloaded to provide various options for instantiating an object. Every class contains at least one constructor, a default constructor, which is automatically created when the programmer does not explicitly provide one. But, programmers often find it useful to declare multiple versions of a class constructor.

Here is an example, if we have a class called `Computer`, we might want to create a computer object for a laptop and another object for a desktop computer. The laptop object might only have the brand name, but the desktop computer might include the brand name and information about the monitor, keyboard and mouse. Listing 13.4 shows the code for a `Computer` class with two overloaded constructors, one for a laptop and one for a desktop computer.

Listing 13.4 Code for overloading a constructor

```

1    public class Computer {
2        String brand, monitor, keyboard, mouse;
3        public Computer(String brand){    // #A
4            this.brand = brand;
5        }
6        public Computer(String brand, String monitor, String keyboard, // #B
7            String mouse){
8            super(brand);    // #C
9            this.monitor = monitor;
10           this.keyboard = keyboard;
11           this.mouse = mouse;
12        }

```

#A Constructor that only takes one String value

#B Constructor that takes four String values

#C This statement calls the first constructor to initialize the brand variable

Now let's take a look at how we can use a class that has overloaded constructors. For this example, I will refer to the Computer class from listing 13.4. The same process applies to overloaded constructors, the calling method decides which constructor to call based on the argument list. In my example, there are two overloaded constructors, one for creating a laptop object and one for creating a desktop object. We can add a third constructor that does not include any arguments, this constructor is called when creating an object that does not include any initial arguments to be passed to the constructor. The no-parameter constructor can be used to create an object based on common values for the instance data. For this example, most of our customers order surface laptops made by Microsoft. So, the no parameter constructor creates an object with the brand value set to Microsoft. Figure 13.6 shows a screenshot of sample output for executing the code in listing 13.5.

```

Microsoft Surface
Macbook Pro
Dell XPS, Samsung Monitor, Logitech Keyboard, Ergonomic Mouse

```

Figure 13.6 Screenshot of output showing three computer objects

Listing 13.5 shows the code with the additional overloaded constructor and a main method that creates three computer instances.

13.5 Complete program used to create three Computer objects

```

1    public class Computer {
2        String brand, monitor, keyboard, mouse;
3        public Computer(){    // #A
4            brand = "Microsoft Surface";
5        }
6        public Computer(String brand){    // #B

```

```

7         this.brand = brand;
8     }
9     public Computer(String brand, String monitor, String keyboard,    //#C
10        String mouse){
11         this.brand = brand;
12         this.monitor = monitor;
13         this.keyboard = keyboard;
14         this.mouse = mouse;
15     }
16     public static void main(String[] args) {
17         Computer laptop1 = new Computer();    //#D
18         System.out.println(laptop1.brand);
19         Computer laptop2 = new Computer("Macbook Pro");    //#E
20         System.out.println(laptop2.brand);
21         Computer desktop = new Computer("Dell XPS", "Samsung Monitor",    //#F
22            "Logitech Keyboard", "Ergonomic Mouse");
23         System.out.println(desktop.brand + ", " + desktop.monitor + ", " +
24            desktop.keyboard + ", "+desktop.mouse);
25     }
26 }

```

#A Overload the default construct with one that sets the default brand to "Microsoft Surface"

#B Overloaded constructor used to create an instance of the Computer class for a laptop

#C Overloaded constructor used to create an instance of the Computer class for a desktop computer

#B Call the overloaded default method

#C Call the overloaded method for a laptop which only takes one String value

#D Call the overloaded method that takes four String arguments

Quick Check 13-4:

For this quick check, use the following code snippet to answer the questions

Listing 13.6 Updated Code for a ToDoList application that includes multiple constructors

```

1     public class ToDoList {
2         String task;
3         int priority;
4         boolean repeat;
5         public ToDoList() {
6             task = "Empty Dishwasher";
7             priority = 1;
8             repeat = true;
9         }
10
11        public ToDoList(String task, int priority, boolean repeat){
12            this.task = task;
13            this.priority = priority;
14            this.repeat = repeat;
15        }
16        public void print(String name){
17            System.out.println(task + ", priority: "+ priority+
18                ", repeat: "+repeat + "\nAssigned to: "+name);
19        }
20        public void print(){
21            System.out.println(task + ", priority: "+ priority+
22                ", repeat: "+repeat);

```

```

23     }
24     public static void main(String[] args){
25         ToDoList todo = new ToDoList("Homework", 1, true);
26         todo.print();
27         todo.print("Bob Fisher");
28     }
29 }

```

1. What is the output if this statement is added to the main method:

```

ToDoList todo2 = new ToDoList();
todo2.print("Casey Damincantonio");

```

- a. Empty Dishwasher, priority: 1, repeat: true
 - b. **No output is produced, this causes an error**
 - c. Empty Dishwasher, priority: 1, repeat: true
Assigned to: Casey Damincantonio
 - d. Homework, priority: 1, repeat: true
2. What is wrong with adding this statement:
- ```

ToDoList todo3 = new ToDoList("Empty Dishwasher", "Bob
Huff");

```
- a. **Nothing, this is correct**
  - b. **The constructor does not take a name and task**
  - c. **There is no overloaded print method for these arguments**
  - d. **The task and the name variables must be switched**

## 13.6 Summary

In this lesson, you learned:

- What it means to overload a method
- Three ways to overload a method
- How to write a class that contains an overloaded method(s)
- How to call the overloaded method(s)
- How to create multiple constructors for a class using method overloading

This lesson reviewed how to write overloaded methods in Java. An overloaded method is useful when your class requires a method that can have different arguments in the parameter list. For code readability, it is helpful to use the same method name, but the parameter lists must be different. For example, a method used to calculate the average of three numbers might have one version that accepts all integer values and a second version that has all double values as arguments. In the next lesson, I introduce another topic related to methods, overriding methods.

**Try this:**

When shopping for televisions, you have many options. Some TVs are considered Smart TVs since they come with Internet access and sometimes they even have apps preloaded. Create a class called `Television` that has at least two constructors, a no parameter constructor for TVs without Internet and a second overloaded constructor for TVs with Internet features. The smart TV constructor should include two additional boolean variables: `hasEthernet` and `hasWiFi`. Some smart TVs have both Ethernet and WiFi. Test your class by creating both types of TV objects. Figure 13.7 provides an example of the output for this class.

```
This is not a smart tv:
Brand: Samsung
Screen Size: 19.0 inches
This IS a smart tv:
Brand: Samsung
Screen Size: 50.5 inches
```

Figure 13.7 Screenshot of sample output for *Try this* activity

**Quick Check 13-1 Solutions:**

- Given the method signature, `public int getPageNumbers()`, which method signature does **not** correctly overload this method:
  - `public int getPageNumbers(String title)`
  - `public void getPageNumbers(int num)`
  - `public double getPageNumbers()`**
  - `public int getPageNumbers(int num)`
- Given the method signature, `public double getAverage(int num1, int num2, int num3)`, which method signature(s) correctly overload this method:
  - `public double getAverage(double num1, int num2, int num3)`
  - `public double getAverage(double num1, double num2, double num3)`
  - `public double getAverage(int num1, double num2, int num3)`
  - All of the above**

**Quick Check 13-2 Solutions:**

- Given the following class:

```
class Student {
 public void printStudent(int studentNum, String fname, String
 lname, double GPA) {
 //code to print student info goes here
 }
 public void printStudent(int studentNum, String name, String
 studentMajor, double GPA) {
```

```

 //code to print student info goes here
 }
}

```

Why won't this code compile?

- a. It does compile, there are no errors
  - b. Both methods cannot have a void return type
  - c. Both methods cannot have the same number of arguments
  - d. The second method does not overload the first since the data types are all the same**
2. Using the code from the first problem, which statement correctly provides the start of an overloaded method:
- a. `public String printStudent(int studentNum, String fname, String lname, double GPA)`
  - b. `public void print(int studentNum, String fname, String lname, double GPA)`
  - c. `public void printStudent()`
  - d. `public String print()`

### Quick Check 13-3:

For this quick check, use the following code snippet to answer the questions

#### Listing 13.3 Code for a ToDoList application

```

1 public class ToDoList {
2 String task;
3 int priority;
4 boolean repeat;
5 public ToDoList(String task, int priority, boolean repeat){
6 this.task = task;
7 this.priority = priority;
8 this.repeat = repeat;
9 }
10 public void print(String name){
11 System.out.println(task + ", priority: "+ priority+
12 ", repeat: "+repeat + "\nAssigned to: "+name);
13 }
14 public void print(){
15 System.out.println(task + ", priority: "+ priority+
16 ", repeat: "+repeat);
17 }
18 public static void main(String[] args){
19 ToDoList todo = new ToDoList("Homework", 1, true);
20 todo.print();
21 todo.print("Bob Fisher");
22 }
23 }

```

1. What is the output if this statement is added to the main method:

```
todo.print("Peggy Fisher");
```

- a. Homework, priority: 1, repeat: true  
Assigned to: Peggy Fisher
  - b. No output is produced, this causes an error
  - c. Homework, priority: 1, repeat: true
  - d. Homework, priority: 1, repeat: false
2. Which line(s) of code are considered overloaded methods?
- a. 5, 10, 14
  - b. 10, 14
  - c. 5
  - d. 10
3. What is wrong with adding this statement: `todo.print("Mow the lawn", "Peggy", 5, true);`
- a. Nothing, this is correct
  - b. The constructor does not take a name and task
  - c. **There is no overloaded print method for these arguments**
  - d. Mow the lawn must be switched with the name Peggy

### Quick Check 13-4 Solution:

#### Listing 13.6 Updated Code for a ToDoList application that includes multiple constructors

```

1 public class ToDoList {
2 String task;
3 int priority;
4 boolean repeat;
5 public ToDoList() {
6 task = "Empty Dishwasher";
7 priority = 1;
8 repeat = true;
9 }
10
11 public ToDoList(String task, int priority, boolean repeat){
12 this.task = task;
13 this.priority = priority;
14 this.repeat = repeat;
15 }
16 public void print(String name){
17 System.out.println(task + ", priority: " + priority+
18 ", repeat: "+repeat + "\nAssigned to: "+name);
19 }
20 public void print(){
21 System.out.println(task + ", priority: " + priority+
22 ", repeat: "+repeat);
23 }
24 public static void main(String[] args){
25 ToDoList todo = new ToDoList("Homework", 1, true);
26 todo.print();

```

```

27 todo.print("Bob Fisher");
28 }
29 }

```

1. What is the output if this statement is added to the main method:

```

ToDoList todo2 = new ToDoList();
todo2.print("Casey Damincantonio");

```

- a. Empty Dishwasher, priority: 1, repeat: true
- b. **No output is produced, this causes an error**
- c. Empty Dishwasher, priority: 1, repeat: true  
Assigned to: Casey Damincantonio
- d. Homework, priority: 1, repeat: true

2. What is wrong with adding this statement:

```

ToDoList todo3 = new ToDoList("Empty Dishwasher", "Bob
 Huff");

```

- a. Nothing, this is correct
- b. **There is no constructor that takes two String arguments**
- c. There is no overloaded print method for these arguments
- d. The task and the name variables must be switched

### **Solution to the Try This activity:**

#### **Listing 13.6 Solution to Try This Activity**

```

1 public class Television {
2 boolean hasEthernet, hasWiFi;
3 String brand;
4 double screenSize;
5 public Television() { // #A
6 hasEthernet = false;
7 hasWiFi = false;
8 brand = "Samsung";
9 screenSize = 19;
10 }
11 public Television(boolean hasEthernet, boolean hasWiFi, String brand,
12 double screenSize){ // #B
13 this.hasEthernet = hasEthernet;
14 this.hasWiFi = hasWiFi;
15 this.brand = brand;
16 this.screenSize = screenSize;
17 }
18 public void print(){
19 if(hasEthernet || hasWiFi)
20 System.out.println("This IS a smart tv: ");
21 else
22 System.out.println("This is not a smart tv: ");
23 System.out.println("Brand: "+ brand);

```

```
24 System.out.println("Screen Size: " + screenSize + " inches");
25 }
26 public static void main(String[] args){
27 Television tv = new Television(); //#C
28 Television smartTV = new Television(false, true, "Samsung", 50.5); //#D
29 tv.print(); //#E
30 smartTV.print(); //#F
31 }
32 }
```

**#A** Overloads the default constructor, provides basic tv data

**#B** Overloads the class constructor to allow for creating Smart TV objects

**#C** Create a standard TV with no internet access

**#D** Create a smart TV object

**#E** Print the information about the regular TV

**#F** Print the information about the smart TV



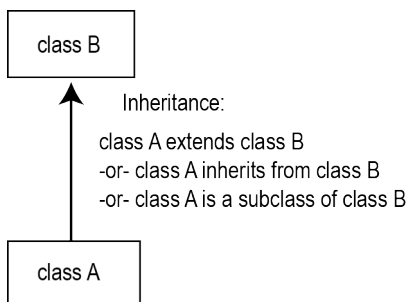
# 14

## Overriding Methods

**After reading lesson 14, you will be able to:**

- Write a class that extends from another class
- Write an overridden method in a subclass
- Use the `super` keyword to reference methods in a parent class
- Use the keyword `instanceof` on objects

This lesson introduces the topic of overriding methods in Java. In Java, a class can extend another class (referred to as the parent class) if we want the classes to share some or all of the data and methods in the parent class. This is called inheritance. Figure 14.1 is a diagram showing inheritance where class B is the parent class to class A. The extended class is considered a subclass or a child class of the parent class.



**Figure 14.1** Inheritance diagram of parent class B and child class A

A method in a child class overrides a similar method in the parent class when it has the exact same method signature. This allows the implementation of the child method to be different than the parent method.

Remember, the method signature includes the return type, the method name, and a parameter list.

#### sample method signature

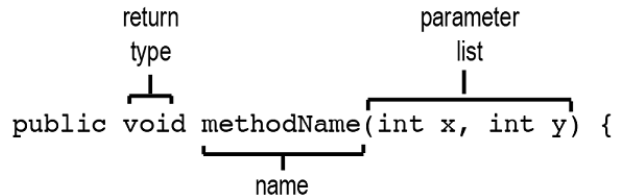


Figure 14.2 Diagram of a method signature

### **Consider This**

You are working around the house and you need a tool. That tool is going to be very different if you are gardening or working on a carpentry project. For example, when gardening, you might be mowing the lawn with a lawn mower. That lawn mower can be operated using gasoline or it can be an old-fashioned reel mower.

But if you are working on a building a deck, the tools might be a circular saw or even something as simple as a hammer and nails.

In this example, tools can be objects that require electric, gas, or operate manually. In other words, power tools vs. manual tools. The term `tool` represents a wide variety of items. When thinking in terms of Java and modelling these items, `tool` can be considered the parent or super class. Then each tool can be represented by a subclass of `tool`. A lawn mower is a `tool`. The term “is-a” is often used to evaluate the relationship between two objects. When this is true, then we have an example of inheritance, a term used in object-oriented programming to represent this parent-child relationship.

The parent `tool` class might have a method to start the tool, ie: `public void start() {}`. But we know that the action of starting a gas powered lawn mower is quite different from a push reel mower. So, we might have two subclasses representing `PowerTools` and `ManualTools`. Each of these subclasses would override the `start` method accordingly.

## 14.1 Write a Class that Extends Another Class

When a class is used to model an object that has variations such as the tool example, it is useful to create an initial class that can be used as a starting point for all objects. Consider a book, if you look around your house or apartment you might see a textbook, a cookbook, or even a novel. All of these items have many things in common, but they are also different. To keep things simple, a book is an object that has a title, author(s), number of pages, and a publisher.

Next, we can define additional classes that share these attributes, but behave slightly different. The pages of a cookbook usually have pictures, a list of ingredients, instructions for combining the ingredients, and instructions for cooking or baking the item. A textbook has

text, images or pictures, and possibly exercises. A novel is usually only text or text and images.

A `Novel`, `Cookbook`, and `Textbook` can all be defined by extending the `Book` class. Each of these additional classes are called subclasses of the `Book` class. This is often referred to as a parent-child relationship. We can use our “is-a” test, a `Novel` is-a `Book`, a `Cookbook` is-a `Book`, and a `Textbook` is-a `Book`. Figure 14.1 is a diagram of this relationship.

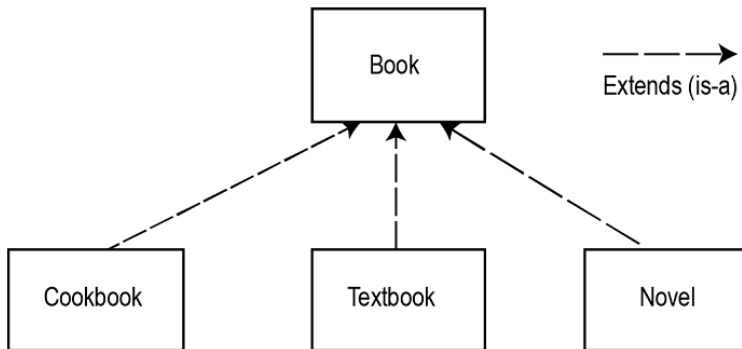


Figure 14.3 Book class and subclasses: Cookbook, Textbook, and Novel

## 14.2 Write an overridden method in a subclass

A subclass within the same package as the parent class can override any method that is not declared private or final. Using the example from figure 14.3, the `Book` class can have a `printPage` method representing the code to print a page of a book. The three subclasses all override this method and print different messages for each instance of the subclass. For this example, I am simply printing a message to demonstrate which version of the `printPage` method is called based on the calling object. Figure 14.4 shows the expected output when executing the code in Listing 14.1, which shows the code for the `Book` class and the three subclasses. Note that each subclass overrides the `printPage` method from the `Book` class.

```

Super class of all books
Print page from textbook: Get Programming with Java

```

Figure 14.4 Sample output demonstrating an overridden method from the `Book` class

### Listing 14.1 Code for the `Book` class and the subclasses `Cookbook`, `Textbook`, and `Novel`

```

1 public class Book {
2 public String title, publisher;
3 public String[] authors;
4 public int numPages;

```

```

5 public Book(String title, String publisher, String authors[],//#A
6 int numPages){
7 this.title = title;
8 this.authors = authors;
9 this.numPages = numPages;
10 this.publisher = publisher;
11 }
12 public void printPage(){ //#B
13 System.out.println("Super class of all books");
14 }
15 public static void main(String[] args) {
16 Book book = new Book("Generic Book", "Manning", new String[]{}, 0); //#C
17 Book text = new Textbook("Get Programming with Java", "Manning", //#D
18 new String[]{"Peg Fisher"},544);
19 book.printPage(); //#E
20 text.printPage(); //#F
21 }
22 }
23 class Textbook extends Book { //#G
24 public Textbook(String title, String publisher, String authors[], int
25 numPages){
26 super(title,publisher,authors,numPages); } //#H
27 public void printPage() { //#I
28 System.out.println("Print page from textbook: "+title); }
29 }
30 class Cookbook extends Book {
31 //code omitted } //#J
32 class Novel extends Book {
33 //code omitted } //#J

```

#A This method is a constructor for the Book class

#B This is a method declared in the parent class that can be overridden by the subclasses

#D Create an instance of the Book class

#D Create an instance of the Book class, but instantiate it with the Textbook class

#E Call the method in the Book class to print a page of the book

#F Call the overridden method on line 19 in the Textbook class, not the super class method

#G Declare a Textbook class that extends the Book class, this is now a subclass of Book

#H Use the keyword super to call the constructor in the Book class, this is reviewed in detail in the next section

#I Override the printPage method in the Book class with a specific Textbook implementation

#J The omitted code looks the same as the Textbook class, but the printPage implementation is specific to each subclass

## The Object Class

It is important to mention that all classes extend from the `Object` class automatically. The `Object` class defines and implements methods common to all classes. Two of the most commonly used methods are `toString()` and `equals()`. The ability to always inherit the `toString()` method enables our programs to include an instance of any object in a print statement, but there is a catch. Have you ever added an object to a print statement and the result was something like this: `Book@7852e922`. The `Object` class defines the `toString()` method to return a string consisting of the name of the class, in our case `Book`, followed by the at-sign character `@`, and the unsigned hexadecimal representation of the hash code of the object.

This is normally not ideal, this is similar to what you will see if you try to print an `Array` or `ArrayList` since both of these data types are objects.

So, to avoid this from happening, we can override the `toString()` method when we create new classes. Remember, all classes extend the `Object` class, but we can provide an overridden `toString()` method to make the output more meaningful. For example, a possible `toString()` method for our `Book` class might print the title and publisher, here is the sample output:

```
Title: Get Programming with Java
Publisher: Manning
```

Here is the overridden `toString()` method:

```
@Override
public String toString(){
 return "Title: "+title+" \nPublisher: "+publisher;
}
```

More information on the Java `Object` class can be found in the Oracle Docs, simply search Java `Object Class`.

<https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html>

### Quick Check 14-1:

- Using the code in Listing 14-1, what output is produced if these statements are added to the main method:

```
Cookbook book = new Cookbook("Amish Cooking", "Manning", new String[]
 {"Fronney Yoder", "Manasses Yoder"}, 300);
book.printPage();
```

- Print page from textbook: Get Programming with Java
  - Print page from textbook: Amish Cooking
  - Print page from cookbook: Amish Cooking
  - Print page from cookbook: Get Programming with Java
- Which line(s) of code could be added to print the author names in addition to the book title?
    - `for(String a:authors) {System.out.print(a + " ");}`
    - `System.out.print(a);`
    - The author names cannot be printed since they are in a `String` array
    - `for(String a:authors) {System.out.print(authors + " ");}`
  - Assume this code is added to the `printPage` method in the `Book` class only:

```
for(String a:authors) {System.out.print(a + " ");} System.out.println("");
```

What is the expected output from executing these two lines of code in the main method:

```
Book text = new Textbook("Get Programming with Java", "Manning",
 new String[]{"Peg Fisher"},544);
text.printPage();
```

- a. Print page from textbook: Get Programming with Java  
Peg Fisher
  - b. Super class of all books  
Peg Fisher
  - c. Print page from textbook: Get Programming with Java
  - d. **Nothing prints since there is an error**
4. The code below is intended to extend the `Book` class and override the `printPage` method. What is wrong with this code:

#### Listing 14.2 Code to extend the Book class

```
class Autobiography extends Book {
 public Autobiography (String title, String publisher, String authors[],
int numPages){
 super(title,publisher,authors,numPages);
 }
 public void printPage(int Year) {
 System.out.println("Print page from autobiography, "+year);
 }
}
```

- a. **Nothing, this code correctly extends the `Book` class and overrides the `printPage` method**
- b. **The overridden method cannot have any arguments in the argument list**
- c. **The class name, `Autobiography`, is too long**
- d. **The `println` statement in the `printPage` method cannot include a variable**

### 14.3 Using the keyword `super`

In the previous example, each of the subclasses of `Book` uses the keyword `super`. When a class extends another class, it automatically inherits access to all public and protected instance data and methods of the super class, including the class constructor. The constructor is the method used to create an object and provide values for the instance data.

Remember, a class that is extended by other classes is considered the super class. So, in our subclass, we can refer to the parent constructor using the keyword `super`. This keyword refers to a method in the parent class and it is used so we don't have to write the same method twice, once in the parent class and again in the child class. In the `Book` example, each subclass refers to the constructor in the `Book` class using the keyword `super` followed by a list of argument values that are required for the `Book` constructor.

Figure 14-5 shows how the `Cookbook` class uses the `Book` constructor to initialize the values for the book title, publisher, author(s), and the number of pages. Notice that the instance data

is only defined in the Book class and it is defined as public. This allows the subclasses to access all of these data fields directly.

```

public class Book {
 public String title, publisher;
 public String[] authors;
 public int numPages;
 public Book(String title, String publisher,
 String authors[], int numPages){
 this.title = title;
 this.authors = authors;
 this.numPages = numPages;
 this.publisher = publisher;
 }
 //Code omitted
 public static void main(String[] args) {
 //Code omitted
 }
}

class Cookbook extends Book {
 public Cookbook(String title, String publisher,
 String authors[], int numPages){
 super(title,publisher,authors,numPages);
 }
 String imagePlaceholder;
 public void printPage() {
 System.out.println("Print page from cookbook: "
 + title);
 }
}

```

*this statement calls the Book constructor*

Figure 14-5 Diagram depicting the use of the super keyword to invoke a method in the parent class from a subclass

### Quick Check 14-2:

1. It is possible for another class to extend our subclasses. For example, here is code to create an Autobiography class that extends the Novel class:

```

class Autobiography extends Novel {
 public Autobiography (String title, String publisher, String authors[],
 int numPages, int year){
 super(title, publisher, authors[],numPages, year);
 }
 public void printPage() {
 System.out.println("Print page from autobiography");
 }
}

```

```
}

```

Why won't this code compile?

- a. The call to `super` must have the same argument list as the `Novel` class constructor
- b. The call to `super` must have the same argument list as the `Book` class constructor
- c. A new variable cannot be introduced in the argument list
- d. The order of the argument list is not correct, the new value must be at the beginning

2. Which code corrects the error in the prior example:

- a. 

```
public Autobiography (String title, String publisher, String authors[],
 int numPages, int year){
 super(year, title, publisher, authors[], numPages);
}
```
- b. 

```
public Autobiography (String title, String publisher, String authors[],
 int numPages, int year){
 super(title, publisher, authors[],numPages);
 this.year = year;
}
```
- c. 

```
public Autobiography (String title, String publisher, String authors[],
 int numPages, int year){
 super(String title, String publisher, String[] authors, int numPages);
 this.year = year;
}
```
- d. 

```
public Autobiography (String title, String publisher, String authors[],
 int numPages){
 super(title, publisher, authors[], numPages, year);
}
```

## 14.4 Using the keyword `instanceof`

In this lesson, I introduced the topic of overriding a method. To override a method, a class must extend a parent class. When there are multiple classes and they are used to extend other classes, it can be confusing to keep track of the objects and how they were instantiated. The keyword `instanceof` can be used to test whether the object is an instance of a class or subclass.

Figure 14.6 shows the output from executing the code in Listing 14.3. In this example, it shows how the `cookbook` object is a type of `Book`. But, `book` is not an instance of `Cookbook` class.



```
Cookbook is a type of Book
Book is not a type of Cookbook
```

Figure 14.6 Sample output from executing the code in Listing 14.3

Listing 14.3 shows only the code to test the instance variables of the classes Book and Cookbook where the Cookbook extends the Book class.

### 14.3 Example of how to use the keyword instanceof

```
1 public class Book {
... //Code for the Book class and Cookbook class are not shown here
22 public static void main(String[] args) {
23 Book book = new Book("Get Programming with Java", "Manning",
24 new String[]{"Peg Fisher"},544);
25 Book cookbook = new Cookbook ("Amish Recipes", "Manning",
26 new String[]{"Frony Yoder"}, 145);
27 if(cookbook instanceof Book) ##A
28 System.out.println("Cookbook is a type of Book");
29 if(book instanceof Cookbook) ##B
30 System.out.println("Book is a type of Cookbook")
31 else
32 System.out.println("Book is not a type of Cookbook");
33 }
34 }
```

#A This conditional statement checks if the object cookbook is an instance of Book, which is true

#B This conditional statement checks the opposite to see if the book object is an instance of Cookbook, which is false

### Quick Check 14-3:

For this quick check, use the following code snippet to answer the questions

### Listing 14.3 Code for a Tool class and a Mower subclass

```
1 public class Tool {
2 public String toolName;
3 public boolean isManual;
4 public Tool(String toolName, boolean isManual) {
5 this.toolName = toolName;
6 this.isManual = isManual;
7 }
8 public void start(){
9 System.out.println("Start the tool");
10 }
11 public static void main(String[] args){
12 Tool tool = new Tool("Rake", true);
13 Tool mower = new Mower("Mower", false, "Lawnboy");
14 Mower reelMower = new Mower("Reel Mower", true, "Fiskars");
15 }
16 }
17 class Mower extends Tool {
18 String brand;
```

```

19 public Mower(String toolName, boolean isManual, String brand){
20 super(toolName, isManual);
21 this.brand = brand;
22 }
23 public void start(){
24 if(isManual) System.out.println("Start pushing");
25 else System.out.println("Move switch to Choke, pull on cord to
start");
26 }
27 }

```

- Using the code in Listing 14-3, what output is produced if these statements are added to the main method:

```

if(tool instanceof Tool) System.out.println("This tool is an instance of the
Tool
class");
else System.out.println("This tool is NOT an instance of the Tool class");

```

- This tool is NOT an instance of the Tool class
  - No output is produced, this causes an error
  - This too is an instance of the Tool class
  - Move switch to Choke, pull on cord to start
- Which line(s) of code could be added to test if the mower object is an instance of the Tool class?
    - if(mower instanceof Tool)
    - if(Mower instanceof Tool)
    - if(mower instanceof Mower)
    - if(reelMower instance of Mower)
  - Which line(s) of code could be added to test if the reelMower object is an instance of the Tool class?
    - if(reelMower instanceof Mower)
    - if(reelMower instanceof Tool)
    - if(tool instanceof reelMower)
    - if(mower instance of reelMower)

## 14.5 Summary

In this lesson, you learned:

- How to write a class that extends from another class
- To write and call a method that is overridden in a subclass
- The use of the `super` keyword
- How to use the keyword `instanceof`

This lesson introduced how to write methods that override other methods in Java. An overridden method is useful when your application requires a parent class with subclasses that

might behave differently from the parent. In addition to this scenario, this lesson also reviewed using the keywords `super` and `instanceof` when working with parent-child classes. In the next lesson, I will introduce another topic related to inheritance called polymorphism. Polymorphism is closely related to what we learned here on overriding methods in subclasses.

### Try this:

Create an application that models Plants. The `Plant` class has two subclasses, `IndoorPlant` and `OutdoorPlant`. All plants need water and sunlight, but outdoor plants need to be covered if the outside temperature is less than 32 degrees Fahrenheit. Figure 14.7 shows sample output from this application:

```
Parent version of careForPlants
Water the plants (Daisy)
Place them in the sun
Cover plants, its cold outside
Parent version of careForPlants
Water the plants (Sunflower)
Place them in the sun
Parent version of careForPlants
Water the plants (Spider Plant)
Place them in the sun
```

Figure 14.7 Screenshot of sample out from Try This activity

- Create a parent class called `Plant`
- Write code for two classes that extend the parent class
- Use `super` to call the parent constructor
- Use `instanceof` on a child object
- Create a method called `careForPlants` in the parent class
- Override this method in the child class(es) using the test for temperature in the outdoor plant class

### Quick Check 14-1 Solutions:

1. Using the code in Listing 14-1, what output is produced if these statements are added to the main method:

```
Cookbook book = new Cookbook("Amish Cooking", "Manning", new String[]
 {"Frony Yoder", "Manasses Yoder"}, 300);
book.printPage();
```

- a. Print page from textbook: Get Programming with Java
- b. Print page from textbook: Amish Cooking
- c. Print page from cookbook: Amish Cooking

- d. Print page from cookbook: Get Programming with Java
2. Which line(s) of code could be added to print the author names in addition to the book title?
- `for(String a:authors) {System.out.print(a + " ");}`
  - `System.out.print(a);`
  - The author names cannot be printed since they are in a String array
  - `for(String a:authors) {System.out.print(authors + " ");}`
3. Assume this code is added to the `printPage` method in the `Book` class only:

```
for(String a:authors) {System.out.print(a + " ");} System.out.println("");
```

What is the expected output from executing these two lines of code in the main method:

```
Book text = new Textbook("Get Programming with Java", "Manning",
 new String[]{"Peg Fisher"},544);
text.printPage();
```

- Print page from textbook: Get Programming with Java  
Peg Fisher
  - Super class of all books  
Peg Fisher
  - Print page from textbook: Get Programming with Java
  - Nothing prints since there is an error
4. The code below is intended to extend the `Book` class and override the `printPage` method. What is wrong with this code:

#### Listing 14.2 Code to extend the Book class

```
class Autobiography extends Book {
 public Autobiography (String title, String publisher, String authors[],
int numPages){
 super(title,publisher,authors,numPages);
 }
 public void printPage(int Year) {
 System.out.println("Print page from autobiography, "+year);
 }
}
```

- Nothing, this code correctly extends the `Book` class and overrides the `printPage` method
- The overridden method cannot have any arguments in the argument list
- The class name, `Autobiography`, is too long
- The `println` statement in the `printPage` method cannot include a variable

**Quick Check 14-2 Solutions:**

1. It is possible for another class to extend our subclasses. For example, here is code to create an **Autobiography** class that extends the **Novel** class:

```
class Autobiography extends Novel {
 public Autobiography (String title, String publisher, String authors[],
 int numPages, int year){
 super(title, publisher, authors[],numPages, year);
 }
 public void printPage() {
 System.out.println("Print page from autobiography");
 }
}
```

Why won't this code compile?

- a. **The call to super must have the same argument list as the Novel class constructor**
  - b. The call to super must have the same argument list as the Book class constructor
  - c. A new variable cannot be introduced in the argument list
  - d. The order of the argument list is not correct, the new value must be at the beginning
2. Which code corrects the error in the prior example:
- a. 

```
public Autobiography (String title, String publisher, String authors[],
 int numPages, int year){
 super(year, title, publisher, authors[], numPages);
 }
}
```
  - b. 

```
public Autobiography (String title, String publisher, String authors[],
 int numPages, int year){
 super(title, publisher, authors[],numPages);
 this.year = year;
 }
}
```
  - c. 

```
public Autobiography (String title, String publisher, String authors[],
 int numPages, int year){
 super(String title, String publisher, String[] authors, int numPages);
 this.year = year;
 }
}
```
  - d. 

```
public Autobiography (String title, String publisher, String authors[],
 int numPages){
 super(title, publisher, authors[], numPages, year);
 }
}
```

**Quick Check 14-3 Solutions:**

For this quick check, use the following code snippet to answer the questions

**Listing 14.3 Code for a Tool class and a Mower subclass**

```

1 public class Tool {
2 public String toolName;
3 public boolean isManual;
4 public Tool(String toolName, boolean isManual) {
5 this.toolName = toolName;
6 this.isManual = isManual;
7 }
8 public void start(){
9 System.out.println("Start the tool");
10 }
11 public static void main(String[] args){
12 Tool tool = new Tool("Rake", true);
13 Tool mower = new Mower("Mower", false, "Lawnboy");
14 Mower reelMower = new Mower("Reel Mower", true, "Fiskars");
15 }
16 }
17 class Mower extends Tool {
18 String brand;
19 public Mower(String toolName, boolean isManual, String brand){
20 super(toolName, isManual);
21 this.brand = brand;
22 }
23 public void start(){
24 if(isManual) System.out.println("Start pushing");
25 else System.out.println("Move switch to Choke, pull on cord to
start");
26 }
27 }

```

- Using the code in Listing 14-3, what output is produced if these statements are added to the main method:

```

if(tool instanceof Tool) System.out.println("This tool is an instance of the
Tool class");
else System.out.println("This tool is NOT an instance of the Tool class");

```

- This tool is NOT an instance of the Tool class
  - No output is produced, this causes an error
  - This too is an instance of the Tool class
  - Move switch to Choke, pull on cord to start
- Which line(s) of code could be added to test if the mower object is an instance of the Tool class?
    - if(mower instanceof Tool)
    - if(Mower instanceof Tool)
    - if(mower instanceof Mower)
    - if(reelMower instance of Mower)

3. Which line(s) of code could be added to test if the `reelMower` object is an instance of the `Tool` class?

- a. `if(reelMower instanceof Mower)`
- b. `if(reelMower instanceof Tool)`**
- c. `if(tool instanceof reelMower)`
- d. `if(mower instance of reelMower)`

**Solution to the Try This activity:**

#### Listing 14.4 Solution to Try This Activity

```

1 public class Plant { //#A
2 public String plantName;
3 public boolean indoorPlant = true;
4 public Plant(String name, boolean indoorPlant){ //#B
5 this.plantName = name;
6 this.indoorPlant = indoorPlant;
7 }
8 public void careForPlants(){ //#C
9 System.out.println("Parent version of careForPlants");
10 System.out.println("Water the plants (" + plantName + ")");
11 System.out.println("Place them in the sun");
12 }
13 public static void main(String[] args){
14 Plant sunflower = new OutdoorPlant("Sunflower"); //#D
15 Plant spiderPlant = new IndoorPlant("Spider Plant"); //#E
16 OutdoorPlant daisy = new OutdoorPlant("Daisy"); //#F
17 if(daisy instanceof OutdoorPlant) //#G
18 daisy.careForPlants(30); //#H
19 sunflower.careForPlants(); //#I
20 spiderPlant.careForPlants(); //#J
21 }
22 }
23 class IndoorPlant extends Plant { //#K
24 public IndoorPlant(String name){
25 super(name, true); //#L
26 }
27 }
28 class OutdoorPlant extends Plant { //#K
29 public OutdoorPlant(String name){
30 super(name, false); //#L
31 }
32 public void careForPlants(double temp){ //#M
33 super.careForPlants(); //#N
34 if(temp < 32)
35 System.out.println("Cover plants, its cold outside");
36 }
37 }

```

**#A** The parent class is called `Plant`

**#B** Constructor for `Plant`, which has two instance variables: plant name and a boolean value (when true is an indoor plant)

**#C** This is the method that will be overridden by child classes

**#D** Create a sunflower object of type `OutdoorPlant`

**#E** Create a spiderPlant object of type `IndoorPlant`

**#F** Create another `OutdoorPlant` object called `daisy`  
**#G** Check if `Daisy` is an `OutdoorPlant`  
**#H** If it is, then call the overridden method with a value for the outside temperature  
**#I** Call the method `careForPlants`, notice there is no value for the temperature so it uses the parent method of `careForPlants`  
**#J** Call the method `careForPlants`, notice it is not overridden for the indoor plants, so the parent method is called again  
**#K** Use `extends` to inherit the public variables and methods from the parent class `Plant`  
**#L** Use `super` to call the constructor from the `Plant` class  
**#M** Override the `careForPlants` method in the subclass `OutdoorPlant`  
**#N** Use `super` to invoke the `careForPlants` method in the `Plant` class and then add additional required logic for the `OutdoorPlant` version of the method.



# 15

## Polymorphism Explained

### After reading lesson 15, you will be able to:

- Understand the definition of polymorphism as it relates to Java programming
- See the benefits of using polymorphism
- Understand the difference between compile time and runtime polymorphism

This lesson introduces the topic of polymorphism in Java, the benefits of using polymorphism and the difference between the two types of polymorphism: compile time and runtime polymorphism.

The term polymorphism originated as a way to refer to a biology topic where a species can have many different forms or stages. In programming, instead of referring to species, we refer to classes and how objects defined by different classes can respond to the same method differently. Here is a simple example, if you are in a room with a few other people and someone says to everyone, “move”, what would happen? Some people might sit if they were standing, people sitting might stand, some people might take a step forward. Different people are told the same thing, but they behave differently.

### Consider This

A track and field coach usually has a team of students who participate in many different types of track and field events. If the coach had three team members, a runner, a javelin thrower, and a pole vaulter together and he shouted “Ready, Set, Go”, each team member would start their event.

The runner would start to run, the javelin thrower would throw the javelin, and the pole vaulter would attempt to vault over the bar.

These three students can be defined by a generic Person class. But each team member would also be a subclass: Runner, JavelinThrower, PoleVaulter. The subclasses would each extend the Person class. Then the “Ready, Set, Go” method would be overridden for each type of athlete.

This example shows how objects defined by different classes can behave differently when invoking the same method call, in this case, the statement “Ready, Set, Go”.

## 15.1 Understand the Definition of Polymorphism

Polymorphism is an important and powerful concept in object-oriented programming. Polymorphism can occur at compile time and at runtime. When a single class contains overloaded methods (which are discussed in Lesson 13), then the same method name is used but the parameter list must be different. In this situation, the same method name performs differently depending on the argument list that is passed to the method. This is considered compile time polymorphism since the compiler identifies which version of the method to call based on the argument list.

The second type is called runtime polymorphism. This type of polymorphism is used to bind a method call at runtime instead of compile time. This polymorphism occurs using overridden methods which are introduced in Lesson 14. In this type, the Java Virtual Machine, or JVM, determines which version of the method to call at runtime. The exact same method signature is used in multiple classes where one class must extend the other class using inheritance. The subclass overrides the method in the superclass and provides different implementations of the same method. Multiple objects which are instances of subclasses of the same superclass can each perform different operations when the method is called. The JVM then decides whether to use the method in the subclass or the method in the superclass at runtime. This lesson demonstrates how the JVM can decide which method to call at runtime.

Let's start by looking at an example. If we have these three classes: `Television`, `Computer`, and `GamingSystem`, we know that each class is a type of `Electronic` device and therefore can be described as a subclass of the superclass `Electronic`.

The three subclasses inherit all the public/protected data and methods from the superclass. Each of the subclass devices contain additional information specific to that device. So, when we want to print the information about each electronic device we need to use the overridden print method in the subclass to ensure we are getting all the specific information for that device. In this case, there is a print method in the `Electronic` class, but each subclass has the same print method signature, but it behaves differently. If we created three `Electronic` objects and instantiated them as a `Television` object, a `GamingSystem` object, and a `Computer` object and then invoked the print method for each reference, the output might look like this:

```
Television
Make: Samsung
Model: Smart
Wifi: true
Size: 60 inches

Gaming System
Make: Sony
Model: 3
Console: Playstation

Computer
Make: Dell
```

```

Model: XP
Type: Laptop
Size: 19 inches

```

**Figure 15.1** screenshot of sample output using the overridden print method in each subclass

Notice that each type of electronic device has some variation of instance data. The television has an indicator for whether it is wifi capable, the gaming system has the console type, and the computer has an indicator for laptop vs. desktop. The superclass, `Electronic`, has a `print` method, and each subclass overrides that method with a custom implementation. Then at runtime the JVM invokes the appropriate `print` method based on the object reference type. This is how polymorphism works. So, each object is told to 'print', but each object has a different version of the `print` method.

### **Quick Check 15-1:**

- 1 Which statement is **NOT** true about runtime polymorphism:
  - a) Polymorphism allows different objects to behave differently when calling the same method name
  - b) Polymorphism uses overridden methods
  - c) Polymorphism is the same as inheritance

## **15.2 Benefits of Using Polymorphism**

You might be wondering why we would use this approach in our applications. Here are some of the benefits of using polymorphism:

- A single reference variable can be used to reference objects of multiple data types
- Polymorphism allows a subclass to use the more general definitions provided by a superclass and override methods to provide specialized method implementations specific to the subclass
- Since runtime polymorphism requires subclasses that override the superclass methods with alternative implementations, this enables code reuse of existing classes without the need to recompile
- Polymorphism enables an object behavior to be determined at runtime
- Compile time polymorphism occurs when a class has method overloading, this is often used for constructors which allows different ways to instantiate an object

### **Quick Check 15-2 Solution:**

- 1 Which of the following is **NOT** a benefit of polymorphism:
  - a) It increases code reuse
  - b) It enables dynamic binding at runtime
  - c) Allows methods that perform similar functions to have the same name

d) It provides access to the private data fields in the superclass

## 15.3 The Difference Between Compile Time and Runtime Polymorphism

There are two types of polymorphism, method overloading and method overriding. This section discusses the difference between the two types. To start, method overloading (Lesson 13) is an example of static polymorphism and it occurs at compile time. Method overriding (Lesson 14) is considered dynamic binding and it occurs at run-time.

From Lesson 13, we know that there are times when we need to write a method that performs the same logic but might have different input. A simple example is a method that finds the average of three numbers. We might want to allow the three numbers to be integers, but there are times when they might all be doubles. The logic inside the method is going to add the numbers, divide by three, and return the result. So, the method signature is very similar except for the type of parameters in the parameter list. In this code snippet, there are two overloaded methods for `findAverage`, and the main method calls the appropriate method based on the arguments in the method call (A and B)

```
public static double findAverage(int a, int b, int c){ // #C
 return (a + b + c)/3.0;
}
public static double findAverage(double a, double b, double c){ // #D
 return (a + b + c)/3.0;
}
public static void main(String[] args) {
 double averageInt = findAverage(10, 23, 45); // #A

 double averageDouble = findAverage(15.5, 14.75, 13.0); // #B
}
```

#A this statement calls the overloaded method with three int parameters

#B this statement calls the overloaded method with three double parameters

#C this method finds the average of three int values

#D this method finds the average of three double values

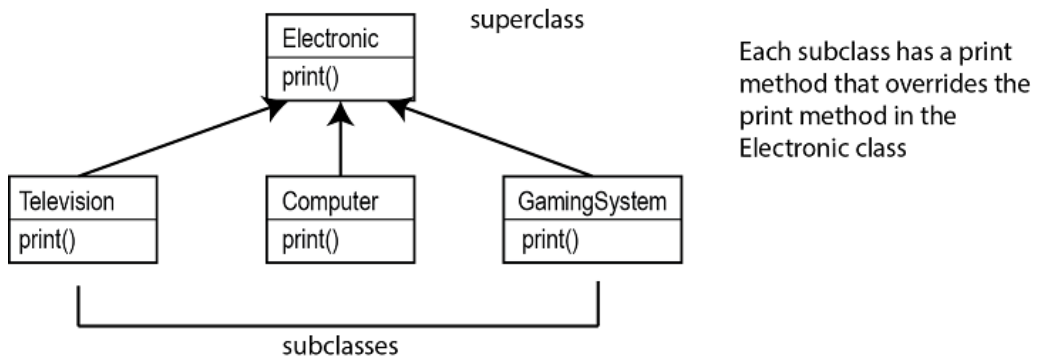
Figure 15.2 screenshot of a code snippet with two overloaded methods

These methods are considered overloaded since they have the same name but different parameter lists. The compiler checks to make sure the method signature is not the same. Since the list of parameters is different, the compiler can check any statement that calls the `findAverage` method to determine if the argument list matches the first method parameter list (double, double, double) or the second parameter list (int, int, int). As long as the method call matches one of these two methods, it will **not** generate an error and the correct method will be invoked. This is a type of polymorphism (where two methods with the same name

behave differently) that is checked at compile time. If the parameters were not different, then this would cause a compile time error.

Now let's talk about method overriding, which is checked at runtime. In general terms, when an object is created using a superclass, it can have a reference to a subclass object which has overridden a superclass method. So, the call to the overridden method is determined at runtime since the compiler does not necessarily know whether the variable has a reference type of the superclass or subclass. This process in which a call to the overridden method is resolved at runtime is known as dynamic binding or runtime polymorphism.

Figure 15.1 is a diagram with a superclass called Electronic which has three subclasses. According to this diagram, a Television is an Electronic, a Computer is an Electronic and a GamingSystem is an Electronic. Each class has a print method. Since the name and parameter list (which is empty) are the same, we know that each subclass is overriding the superclass print method.



**Figure 15.2** Hierarchy chart of an Electronic superclass and three subclass: Television, Computer, GamingSystem

Based on this diagram, we can use the following statement to define an Electronic reference variable that is referencing a Television object:

```
Electronic tv = new Television("Samsung", "Smart", "60 inches");
```

Now, if we want to print the television information, we use this statement: `tv.print();` The compiler checks to make sure that Television is a subclass of Electronic and if the subclass has a print method. In this case, it does have a print method. At runtime, the JVM invokes the print method from the Television class and not the method from the Electronic class. This allows the same method name to perform a different action which is the definition of polymorphism. This action is resolved at runtime and not compile time.

### Quick Check 15-3:

- 1 Given the following code snippet, which print method is invoked:

```
Electronic tv = new Television("Samsung", "Smart", "60 inches");
Electronic c = new Computer("Dell", "XP", "Laptop", "19 inches");
Electronic e = new Electronic("Samsung", "4k");
e = tv;
e.print();
```

- a. The print method from the Electronic class
- b. The print method from the Television class
- c. None, the statement `e = tv;` generates a compile time error
- d. None, the statement `e = tv;` generates a runtime error

## 15.4 Summary

In this lesson, you learned:

- The definition of polymorphism in object-oriented programming
- To describe the benefits of using polymorphism
- The difference between compile time and runtime polymorphism

This lesson reviewed the programming definition of polymorphism in Java. This lesson also introduced the benefits of using polymorphism. The last section reviewed the differences between compile time and runtime polymorphism. The next lesson is still on polymorphism where I will walk through a real-life example of how we can use polymorphism.

### ***Try this:***

You have been hired as an intern at a local Bike shop. The shop owner wants you to work on an inventory system. The shop sells the following types of bikes:

- Mountain Bike
- Road Bike
- Racing Bike

All three types of bikes have some common characteristics, such as make, model, color. But then each bike type has some specific characteristics. For example, a racing bike has information about the gear shift. A road bike might need more detailed information about the frame material and weight. I am not an expert in biking, but for this example, each type of bike is unique but also shares some overall characteristics of the other bikes.

Using this information and the following code for a Bike class, create a subclass that overrides the print method for a MountainBike. Add a variable for the seatheight of the mountain bike and include that in the print method. Then create a Bike reference variable that is instantiated as a MountainBike and prints the additional information about the seat height. Finally, using polymorphism, call the print method to print the information about the MountainBike.

**Listing 15.1 Bike Class**

```

1 public class Lesson15_TryThisSolution {
2 public static void main(String[] args) {
3
4 }
5 }
6 class Bike { //#A
7 public String type, make, model;
8 public int gear;
9 public int speed;
10 public Bike(String type, String make, String model) {
11 this.type = type;
12 this.make = make;
13 this.model = model;
14 }
15 public Bike(String type, String make, String model, int gear, int speed) {
16 this.type = type;
17 this.make = make;
18 this.model = model;
19 this.gear = gear;
20 this.speed = speed;
21 }
22 public void print(){ //#B
23 System.out.println("Bike info: " + type + ", " + make + ", "
24 + model);
25 }
26 }

```

#A Declare the Bike class

#B Define a print method

**Quick Check 15-1 Solution:**

- 1 Which statement is **NOT** true about polymorphism:
  - a Polymorphism allows different objects to behave differently when calling the same method
  - b Polymorphism uses overridden methods
  - c Polymorphism is the same as Inheritance**

**Quick Check 15-2 Solution:**

- 1 Which of the following is **NOT** a benefit of polymorphism:
  - a It increases code reuse
  - b It enables dynamic binding at runtime
  - c Allows methods that perform similar functions to have the same name
  - d It provides access to the private data fields in the superclass**

**Quick Check 15-3 Solution:**

1. Given the following code snippet, which print method is invoked:

```
Electronic tv = new Television("Samsung", "Smart", "60 inches");
```

```
Electronic c = new Computer("Dell", "XP", "Laptop", "19 inches");
Electronic e = new Electronic("Samsung", "4k");
e = tv;
e.print();
```

- a. The print method from the Electronic class
- b. The print method from the Television class**
- c. None, the statement `e = tv;` generates a compile time error
- d. None, the statement `e = tv;` generates a runtime error

### **Solution to the Try This activity:**

#### **Listing 15.1 Bike Class with subclass Mountain Bike using an overridden print method**

```
1 public class Lesson15_TryThisSolution {
2 public static void main(String[] args) {
3 Bike b = new MountainBike("Mountain Bike", "REI", "Diamondback", //#A
4 10, 20, 27.5);
5 b.print(); //#B
6 }
7 }
8 class Bike {
9 public String type, make, model;
10 public int gear;
11 public int speed;
12 public Bike(String type, String make, String model) {
13 this.type = type;
14 this.make = make;
15 this.model = model;
16 }
17 public Bike(String type, String make, String model, int gear, int speed) {
18 this.type = type;
19 this.make = make;
20 this.model = model;
21 this.gear = gear;
22 this.speed = speed;
23 }
24 public void print() {
25 System.out.println("Bike info: " + type + ", " + make + ", "
26 + model);
27 }
28 }
29 class MountainBike extends Bike {
30 public double seatHeight;
31 public MountainBike(String type, String make, String model, int gear,
32 int speed, double seatHeight) {
33 super(type, make, model, gear, speed);
34 this.seatHeight = seatHeight;
35 }
36 @Override //#C
37 public void print() {
38 System.out.println("Mountain Bike info: " + type + ", " + make + ", "
39 + model + ", seat height: " + seatHeight);
40 }
41 }
```



**#A** Create a `Bike` reference variable that is instantiated as a `MountainBike` object  
**#B** Call the `print` method using the `MountainBike` object, so this invokes the `print` method on line 37  
**#C** This method overrides the `print` method in the `Bike` class

# 16

## *Polymorphism In Action*

### **After reading lesson 16, you will be able to:**

- Define objects using polymorphism
- Use upcasting and downcasting with objects
- Describe how the virtual machine chooses the correct method when a subclass overrides a superclass method

This lesson shows examples of polymorphism in Java. As I stated previously, there are two types of polymorphism. Method overloading (Lesson 13) is an example of static polymorphism and it occurs at compile time. Method overriding (Lesson 14) is considered dynamic binding and it occurs at run-time.

### **Consider This**

Consider the geometric term, shape. For example, we can say that a circle is a shape, and a rectangle is a shape. But if we want to find the area of these two shapes, the formula is quite different. The area of a circle is  $\pi * \text{radius squared}$ . The area of a rectangle is  $\text{length} * \text{width}$ .

So, in this example, we have a superclass called Shape, and several subclasses. For this example, let's just use Circle and Rectangle. A Circle is-a Shape, a Rectangle is-a Shape so they both pass the inheritance test. There might be some characteristics of all shapes that are included in the Shape class. But we know that each subclass must have a unique method for calculating the area.

If we have our superclass, Shape, and we create subclasses that extend Shape called Circle and Rectangle then we can create objects that behave differently when they invoke the same method name, calculateArea. For example, we can create a Shape object, then we can set the shape variable reference to a new Rectangle object. Here is the sample code:

```
Shape s = new Rectangle();
```

The reference address of the Rectangle object is stored in the shape reference variable. This is valid since a Rectangle is-a Shape. In Java, a reference variable is polymorphic because it can reference objects of a different type from its own, as long as they are subclasses of its type. This is an example of polymorphism.

## 16.1 Setup Classes Using Inheritance to Demonstrate Polymorphism

My neighbor, who used to be the field hockey coach, is now the athletic director for our local high school. In this role, she is in charge of all the sporting events for our school. She also needs to have a current roster of all students that are athletes. This is used to periodically check their GPA to make sure they are still eligible to play sports.

If she asked me to create an application to help her manage all the sports teams and the athletes on these teams, I would start by examining the relationship of these entities. For this example, I am using classes to represent a SportsTeam, Student, Athlete, FootballPlayer, TennisPlayer, FieldHockeyPlayer, and SoccerPlayer. The SportsTeam class will be used to create a team object with a list of all the athletes on that team. The Student class is a superclass since all athletes are students. And each subcategory of Athlete extends the Athlete class, so they are all subclasses of the Athlete class. Figure 16.1 shows the hierarchy of these classes.

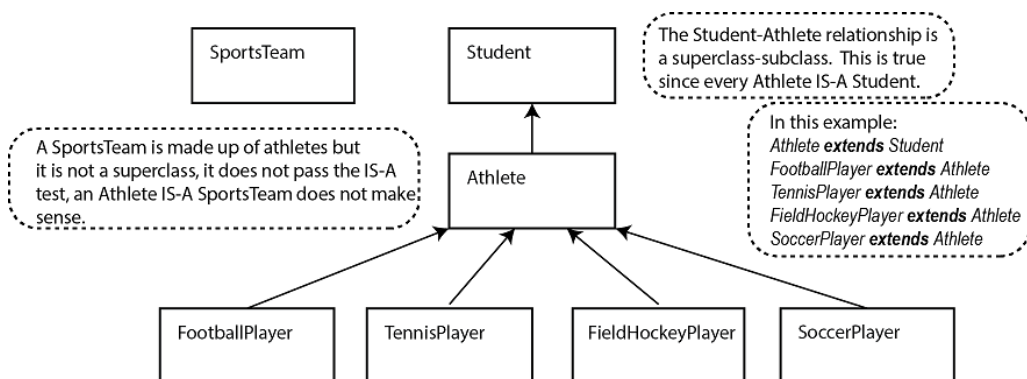


Figure 16.1 Hierarchy diagram for the Sports Teams and Students on these teams

Now that we have our diagram of the classes and their relationships, the next step is to identify the attributes and behaviors for each class. For this example, here are the requirements for each class:

### SportsTeam:

- Type of sport
- Team name
- List of athletes on the team

### Student:

- First and last name
- Grade
- Age
- GPA

### FootballPlayer

- All the athlete information
- Position
- Stats

### TennisPlayer

- All the athlete information
- Singles/doubles
- Stats

**Athlete:**

- All the student information
- Size (S, M, L, XL, XXL)
- Active or inactive
- Eligibility (based on GPA)

**FieldHockeyPlayer**

- All the athlete information
- Position
- Stats

**SoccerPlayer**

- All the athlete information
- Position
- Stats

**Quick Check 16-1:**

1. Using Figure 16.1, which statement is **not** valid:
  - a. An Athlete is a Student
  - b. A TennisPlayer is a Student
  - c. A TennisPlayer is an Athlete
  - d. A Student is an Athlete

## 16.2 Define Objects Using Polymorphism

Using the information from section 16.1, I have started an application that can be used to keep track of all the student athletes. Listings 16.1-16.5 show the code for the Lesson16\_Example (which contains the main method), SportsTeam, Student, Athlete, and the FieldHockeyPlayer classes. To save space, I have omitted the FootballPlayer, TennisPlayer, and SoccerPlayer classes in this section, but the complete code is available at the end of this lesson. For this example, I have decided to place each class in a separate file. The package at the top of each code file is the same for each class, so the compiler knows these classes are all part of one application. Before diving into the code, let's look at figure 16.2. This figure shows the hierarchy of our classes and provides a subset of a UML diagram for each class so you can see the methods in the superclass and the subclasses.

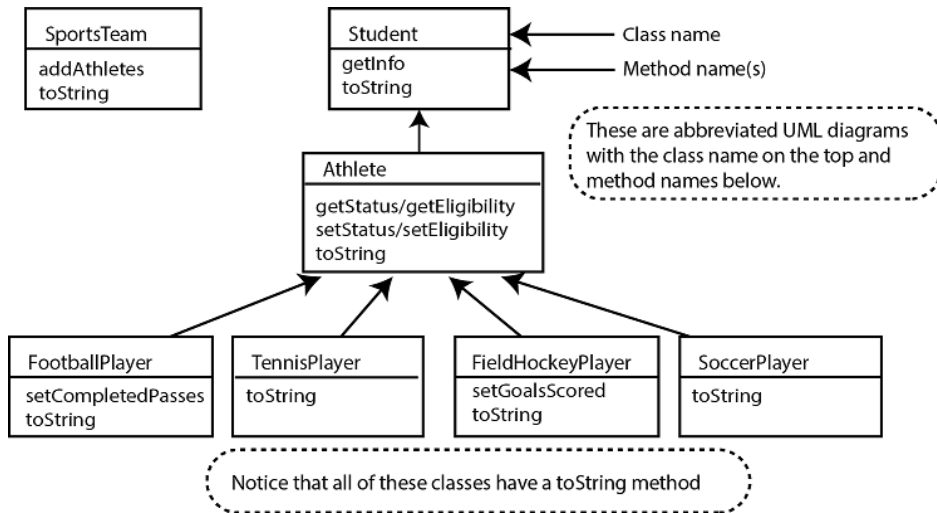


Figure 16.2 Abbreviated UML diagram for the sports team application

### UML Explained

UML diagrams are used to help visualize a class. UML stands for Unified Modeling Language. It is often used to depict a class and includes the class name in the top compartment, followed by a list of attributes in the middle compartment, and a list of methods in the bottom compartment.

Next to each attribute and method, you might see a symbol indicating the visibility, such as:

+public

-private

#protected

Here is an example of a UML diagram:

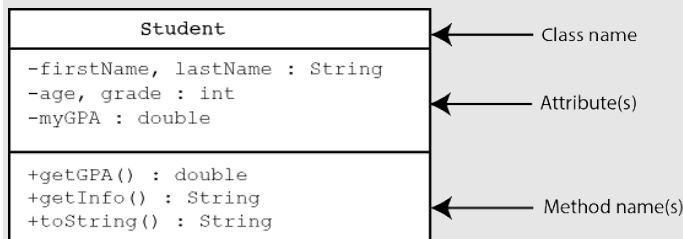


Figure 16.3

The diagram in figure 16.2 shows each class and the methods defined in each class. It is especially important to note that each class has its own version of a `toString` method. Since the **Athlete** class extends the **Student** class, this is an example of overriding a method. The same is true for each of the subclasses under **Athlete**, they each override the `toString` method

in the Athlete class. Examples of how this works is included in the next section. First, let's see the code for each of these classes, then I will point out how polymorphism is used in this example.

### Listing 16.1 Lesson16\_Example class

```

1 public class Lesson16_Example {
2 public static void main(String[] args) {
3 int fhCount = 0;
4 Student fh1 = new FieldHockeyPlayer("Tish", "Maclay", 16, 11,
5 3.9, true, "M", "GoalKeeper"); //A
6 Student fp1 = new FootballPlayer("Darryl", "McKnight", 17, 12,
7 1.5, true, "XL", "Quarterback"); //A
8 Student tp1 = new TennisPlayer("Abhiram", "Nirman", 17, 11, 4.0,
9 true, "L", true); //A
10 Student fh2 = new FieldHockeyPlayer("Bethan", "Palmer", 15, 10,
11 3.5, true, "S", "Midfield"); //A
12 fh2.setGoalsScored(5); //B
13 Student s1 = new Student("Zahraa", "Khalil", 14, 9, 3.95); //C
14 FieldHockeyPlayer[] fhPlayers = new FieldHockeyPlayer[10]; //D
15 Student[] students = new Student[50]; //E
16 students[0] = fh1; //F
17 students[1] = fp1; //F
18 students[2] = tp1; //F
19 students[3] = fh2; //F
20 students[4] = s1; //F
21 for (Student s : students) { //G
22 if (s instanceof FieldHockeyPlayer) { //H
23 fhPlayers[fhCount] = (FieldHockeyPlayer) s; //I
24 fhCount++;
25 }
26 }
27 SportsTeam fhTeam = new SportsTeam("Field Hockey", "Huskies"); //J
28 fhTeam.addAthletes(fhPlayers); //K
29 System.out.println(fhTeam.toString()); //L
30 }
31 }

```

**#A** Create four students: (2) field hockey players, 1 football player, 1 tennis player  
**#B** Use the setGoalsScored method for a field hockey player to set the number of goals scored  
**#C** Create a student object  
**#D** Create an array to hold all the field hockey players  
**#E** Create an array to hold the list of all students (remember, each athlete is a student)  
**#F** Add all the students to the Student array  
**#G** Use an enhanced for loop to go through all the students  
**#H** Check if the student is a field hockey player (using the instanceof keyword)  
**#I** If it is a field hockey player, add them to the field hockey array  
**#J** Create a new sports team for the field hockey team  
**#K** Add the field hockey players to the team  
**#L** Print out all players on the field hockey team

In Listing 16.1, we start to see a glimpse of polymorphism. Notice that on lines 16-19, I am adding my athletes to the students array. Since each type of athlete (FootballPlayer, SoccerPlayer, etc.) extends Athlete and the Athlete class extends Student, we can assign each

element in the students array to a reference of a Student, or Athlete, or FieldHockeyPlayer, etc. This is possible since each athlete type is a subclass of the Athlete class and the Athlete class is a subclass of the Student class.

In other words, an object of a subclass can be used anywhere an object of the superclass is used. The JVM decides which method to call at runtime based on the object type, this is considered dynamic binding. So, in the example above, variable fh1 is created as a Student reference type but then it is pointing to a FieldHockeyPlayer object. So, when invoking a method on the object fh1, the JVM first looks to the FieldHockeyPlayer class first, if it is **not** found, it goes to the superclass, in this case the Athlete class. If the Athlete class does **not** have a corresponding method, it goes to the superclass of the Athlete, which is the Student class.

The next listing is for the SportsTeam class.

#### Listing 16.2 SportsTeam class

```

1 package lesson16_example;
2 public class SportsTeam {
3 private String sportType, teamName;
4 private Athlete[] teamMembers = new Athlete[20];
5 private int count = 0;
6 public SportsTeam(String sportType, String teamName) {
7 this.sportType = sportType;
8 this.teamName = teamName;
9 }
10 public void addAthletes(Athlete[] athletes) {
11 for (Athlete a : athletes) {
12 if (!(a == null)) { //#A
13 a.setEligibility(); //#B
14 if (a.getStatus() && a.getEligibility()) { //#C
15 teamMembers[count] = a;
16 count++;
17 }
18 }
19 }
20 }
21 @Override
22 public String toString() { //#D
23 String names = "";
24 for (Athlete a : teamMembers) {
25 if (!(a == null)) {
26 names += a.toString() + "\n"; //#E
27 }
28 }
29 return sportType + " " + teamName + "\n" + names;
30 }
31 }

```

**#A** If the array element is not set to an object, it will be null, so it should not be added to the team list  
**#B** This statement calls the setEligibility method to check the GPA of the student, if it is < 2.0, they are ineligible  
**#C** If the athlete is active and has a GPA of 2.0 or above, they are added to the team list  
**#D** This method overrides the toString method in the Object class  
**#E** Polymorphism is used to determine which toString method to call, in this case, the Athlete.toString() method

The SportsTeam class is designed to create a roster of eligible athletes for each sports team. This class accepts a list of potential athletes in the method starting on line 10. To avoid encountering a null pointer exception, line 12 checks to make sure each value in the athletes' array has been instantiated. Then it checks to see if each athlete is active and eligible, if the athlete is eligible, then they are added to the list of team members. There is also a toString method that prints the team name and a list of athletes on the team. In this example it might seem strange to have the override statement on line 21, but even though the SportsTeam class does not explicitly extend any other classes, remember, all classes extend the Object class by default. So, the toString method in the SportsTeam class overrides the toString method in the Object class. The next listing is for the Student class.

### Listing 16.3 Student class

```

1 package lesson16_example;
2 public class Student {
3 private String firstName, lastName;
4 private int age, grade;
5 private double myGPA;
6 private double myGPA;
7 public Student(String firstName, String lastName, int age, int grade,
8 double gpa) {
9 this.firstName = firstName;
10 this.lastName = lastName;
11 this.age = age;
12 this.grade = grade;
13 this.myGPA = gpa;
14 }
15 public double getGPA() { return myGPA; }
16 @Override // #A
17 public String toString() {
18 String gradeInfo;
19 switch(grade){ // #B
20 case 9: gradeInfo = "Freshman"; break;
21 case 10: gradeInfo = "Sophomore"; break;
22 case 11: gradeInfo = "Junior"; break;
23 case 12: gradeInfo = "Senior"; break;
24 default: gradeInfo = "Invalid year"; break;
25 }
26 return "Student Name = " + firstName + " " + lastName + " " + "Age = " +
27 age + " Year = " + gradeInfo + "\n";
28 }
29 }

```

**#A** Override the toString method in the Object class

**#B** Convert the numeric grade to the corresponding term (Freshman, Sophomore, Junior, Senior)

The Student class is designed to represent all students. It contains information about the student's name, age, grade, and GPA. This class is the superclass to the Athlete class. This makes sense since every Athlete must be a Student. Note that this class has two public methods:

```
public double getGPA()
```



```
public String toString()
```

Since these methods are public, they become accessible to any subclass that extends the Student class.

#### Listing 16.4 Athlete class

```

1 package lesson16_example;
2 public class Athlete extends Student { //#A
3 private boolean active = true, eligible = true;
4 private String size;
5 public Athlete(String fname, String lname, int age, int grade, double gpa,
6 boolean active, String size) {
7 super(fname, lname, age, grade, gpa); //#B
8 this.active = active;
9 this.size = size;
10 }
11 public void setEligibility() {
12 if (this.getGPA() < 2.0) {
13 eligible = false; } }
14 public void setStatus(boolean status) {
15 active = status; }
16 public boolean getStatus() {
17 return active; }
18 public boolean getEligibility() {
19 return eligible; }
20 @Override
21 public String toString() { //#C
22 return super.toString() + "Active: " + active + "\nEligible: " //#D
23 + eligible + "\nUniform Size: " + size + "\n";
24 }
25 }
```

**#A** The Athlete class extends the Student class

**#B** The constructor uses the keyword super to call the Student constructor for the name, age, grade and gpa

**#C** This method overrides the toString method from the Student class

**#D** This method also uses the keyword super to call the toString method in the Student class

Here is our first subclass of the Student class. The athlete class is not specific to a certain sport, it is representative of all students who play sports. It includes access to all the public and protected characteristics of the Student class, and then adds any additional characteristics specific to all athletes. In this example, I have added two boolean variables that are used to indicate if the athlete is active/inactive and eligible/ineligible. In the code listing 16.4 on line 12, you can see that the program is referencing the getGPA method in the Student class. Since that method is defined as a public method, it is available to all the subclasses. This is an example of how inheritance can be used in Java.

The method starting on line 21 is an overridden version of the toString method. So, when an object reference is for an Athlete object, if the code calls the toString method, it will execute this method. Inside this method I am using the keyword super to reference the toString method in the Student class. This enables my program to use the code already written for part of the String that gets returned from a call to this method.

Note that this class has the following public methods:

```
public void setEligibility(boolean)
public void setStatus(boolean)
public boolean getEligibility()
public boolean getStatus
public String toString()
```

Since these methods are public, they become accessible to any subclass that extends the Athlete class in addition to the two public methods from the Student class.

#### Listing 16.5 FieldHockeyPlayer class

```
1 package lesson16_example;
2 public class FieldHockeyPlayer extends Athlete { //#A
3 private String position;
4 private int goalsScored = 0;
5 public FieldHockeyPlayer(String fname, String lname, int age, int grade,
6 double gpa, boolean active, String size, String position) { //#B
7 super(fname, lname, age, grade, gpa, active, size);
8 this.position = position;
9 }
10 public void setGoalsScored(int gs) {
11 goalsScored = gs;
12 }
13 @Override
14 public String toString() { //#C
15 return super.toString() + "Position: "
16 + position + "\nGoals Scored: " + goalsScored;
17 }
18 }
```

#A The FieldHockeyPlayer class extends the Athlete class

#B The constructor uses the keyword super to call the Athlete constructor

#C This method overrides the toString method from the Athlete class

In the FieldHockeyPlayer class, it has access to all of the following methods from the Athlete class and the Student class:

```
public double getGPA()
public void setEligibility(boolean)
public void setStatus(boolean)
public boolean getEligibility()
public boolean getStatus
public String toString()
```

Since the toString method is actually overridden in the FieldHockeyPlayer class, it gets invoked whenever a variable references a FieldHockeyPlayer object. This is the power of polymorphism, here is an example:

#### Listing 16.6 Example of invoking an overridden method using polymorphism

```
1 package lesson16_example;
2 public class Lesson16_Example {
3 public static void main(String[] args) {
```

```

4 Student s = new Student("Tish", "Maclay", 16, 11, 3.95); // #A
5 s = new FieldHockeyPlayer("Tish", "Maclay", 16, 11, // #B
6 3.9, true, "M", "GoalKeeper");
7 s.getGPA(); // #C
8 System.out.println(s.toString()); // #D
9
10 Athlete a = new Athlete("Tish", "Maclay", 16, 11, 3.95, true, "S"); // #E
11 a = new FieldHockeyPlayer("Tish", "Maclay", 16, 11, // #F
12 3.9, true, "M", "GoalKeeper");
13 a.getGPA(); // #G
14 a.getEligibility(); // #H
15 a.getStatus(); // #H
16 a.setEligibility(); // #H
17 a.setStatus(true); // #H
18 System.out.println(a.toString()); // #I
19 }
20 }

```

#A Create a variable to reference a Student object

#B Update the variable reference to now be a FieldHockeyPlayer object (first step in polymorphism)

#C Access the `getGPA()` method from the Student class

#D Access the overridden `toString` method in the FieldHockeyPlayer class (example of polymorphism)

#E Create a variable to reference an Athlete object

#F Update the variable reference to now be a FieldHockeyPlayer object

#G Access the `getGPA()` method from the Student class

#H Access the `getEligibility`, `getStatus`, `setEligibility`, `setStatus` methods from the FieldHockeyPlayer class

#I Access the overridden `toString` method from the FieldHockeyPlayer class (example of polymorphism)

Now that we have all the classes defined, let's review how the virtual machine determines which method to invoke. Remember, if a class extends another class, it can access any public or protected attributes and methods of the superclass. For example, in code listing 16.6, I start by defining a variable that references a Student object. Next, I am going to update the reference value of this variable to a FieldHockeyPlayer data type. At this point, the variable has access to the public met

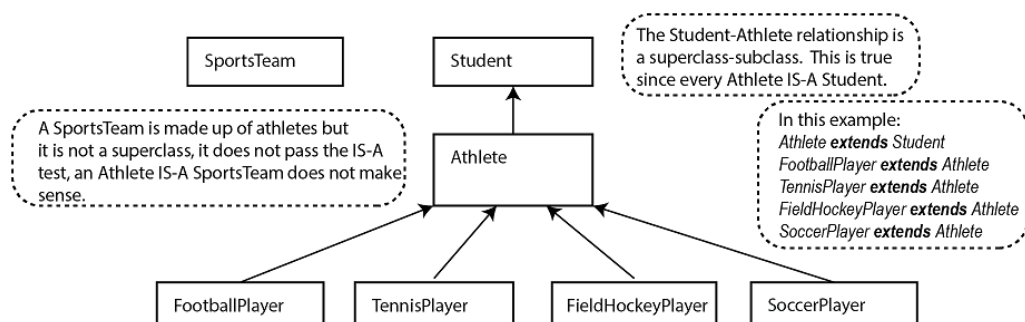


Figure 16.4 Hierarchy diagram for the Sports Teams and Students on these teams (reprinted)

Using the diagram in Figure 16.1, I've created a table with sample code that instantiates several types of objects. Most of the statements are valid, but there is one statement that causes a compile time error: **Incompatible types: Student cannot be converted to Athlete**. Review the table and notice the use of the superclass and subclass references.

**Table 16.1** contains examples of valid/invalid polymorphic objects

| Statement                                          | Valid/Invalid | Description                                                                                                                                                                                                                                                                    |
|----------------------------------------------------|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Student s = new Student ();</code>           | Valid         | The variable <code>s</code> is a reference to a Student object                                                                                                                                                                                                                 |
| <code>Student a = new Athlete();</code>            | Valid         | The <code>a</code> variable points to an Athlete object, but the reference type is a Student (which is allowed since Athlete is a subclass of Student)                                                                                                                         |
| <code>Athlete a = new Student();</code>            | Invalid       | A variable defined as a subclass cannot have a reference to a superclass object                                                                                                                                                                                                |
| <code>Athlete a = new Athlete();</code>            | Valid         | The variable <code>a</code> is a reference to an Athlete object                                                                                                                                                                                                                |
| <code>Athlete a = new TennisPlayer();</code>       | Valid         | The variable <code>a</code> is a reference to an Athlete object, but it can be updated to reference a TennisPlayer object, FootballPlayer object, SoccerPlayer object, or even a FieldHockeyPlayer object because they are all subclasses of the Athlete class                 |
| <code>Student s = new<br/>FootballPlayer();</code> | Valid         | The variable <code>s</code> is a reference to a Student object, but it can be updated to reference an Athlete object, TennisPlayer object, FootballPlayer object, SoccerPlayer object, or even a FieldHockeyPlayer object because they are all subclasses of the Student class |
| <code>SoccerPlayer sp = new Athlete();</code>      | Invalid       | A variable defined to reference a subclass cannot have a reference to a superclass object                                                                                                                                                                                      |

When working with inheritance, it is sometimes difficult to understand what is allowable when using a superclass or subclass to create various objects. A general rule is that the variable declaration on the left-hand-side must be the same or a superclass of the instantiated object on the right-hand-side. The benefit of creating a variable using the parent class is that the variable name can be reassigned to any of the subclasses. But, don't be fooled, since the variable is considered a reference to a parent object, it only has access to public data and methods in the parent class and access to any methods that are overridden in the child class.

The next section discusses how we can work with upcasting and downcasting to get around this limitation.

**Quick Check 16-2:**

Given this code snippet, answer the following questions:

**Listing 16.7 Sample code snippet for the quick check**

```

1 public class Lesson16_Example {
2 public static void main(String[] args) {
3 Student fp1 = new SoccerPlayer("Gita", "Chandra", 16, 11,
4 3.5, true, "XL", "Fullback"); //A
5 Student tp1 = new TennisPlayer("Troy", "Miles", 17, 11, 1.95,
6 true, "L", true); //B
7 Student fh2 = new FieldHockeyPlayer("Cindy", "Bierly", 15, 10,
8 3.5, true, "S", "Midfield"); //C
9 Student s1 = new Student("Marchela", "Bozhilova", 14, 9, 3.95); //D
10 Student[] students = new Student[50];
11 students[0] = fp1;
12 students[1] = fp1;
13 students[2] = tp1;
14 students[3] = fh2;
15 students[4] = s1;
16 TennisPlayer[] tplayers[10];
17 int count = 0;
18 for (Student s : students) {
19 if (s instanceof FieldHockeyPlayer) { //E
20 tplayers[count] = (FieldHockeyPlayer) s; //F
21 count++;
22 }
23 }
24 SportsTeam tennisTeam = new SportsTeam("Tennis", "Huskies");
25 tennisTeam.addAthletes(tplayers);
26 System.out.println(tennisTeam.toString());
27 }
28 }

```

1. Statement at **#A** causes what type of compiler error?
  - a. Incompatible types
  - b. No error, the statement is correct
  - c. Missing arguments
  - d. Syntax error
2. Why won't the student at **#B** be added to the list of tennis players?
  - a. The object has a variable reference to an Athlete object
  - b. The player will be added
  - c. The GPA is less than the threshold to be eligible
3. What needs to change in the statement at **#E** to find only TennisPlayer objects?
  - a. Change instanceof to subclassof
  - b. Change FieldHockeyPlayer to TennisPlayer
  - c. Nothing, the statement is correct

4. What needs to change in the statement at #F?
  - a. Change `tpplayers[count]` to `(TennisPlayer)tpplayers[count]`
  - b. Change `FieldHockeyPlayer` to `TennisPlayer`
  - c. Nothing, the statement is correct

## 16.3 What Does it Mean to Upcast and Downcast objects

Upcast and downcast are two terms associated with polymorphism. In Java, upcasting means casting an object to a superclass type, while downcasting means casting an object to a subclass type. A subclass object that is defined using a superclass is considered upcasting and can be done implicitly. In the code listing 16.8 (which is a reprint of Listing 16.1), lines 16 – 19 are examples of upcasting. This is true since each variable reference on the right-hand-side is from a subclass of the variable reference on the left-hand-side.

Upcasting is always allowed, but we need to be careful with downcasting. When we use downcasting on an object, it can cause a `ClassCastException`. In the code Listing 16.6, I have included a check to make sure that the object is a `FieldHockeyPlayer` before downcasting the reference object in the array. This is done using the `instanceof` keyword.

In order to downcast, we must explicitly use the subclass name and cast the object to that subclass. Line 23 shows an example of how to downcast from a `Student` reference to a `FieldHockeyPlayer` reference. This is necessary for this code since I wanted to add field hockey athletes to the list of field hockey players. An array is created to hold the list of field hockey players on line 14. So, every object added to this array must be of type `FieldHockeyPlayer` (or a subclass of `FieldHockeyPlayer` if there was one).

### Listing 16.8 Examples of Upcasting and Downcasting

```

1 public class Lesson16_Example {
2 public static void main(String[] args) {
3 //code omitted - see Listing 16.1
4 FieldHockeyPlayer[] fhPlayers = new FieldHockeyPlayer[10];
5 Student[] students = new Student[5];
6 students[0] = fh1; ##A
7 students[1] = fp1;
8 students[2] = tp1;
9 students[3] = fh2;
10 students[4] = s1;
11 for (Student s : students) {
12 if (s instanceof FieldHockeyPlayer) {
13 fhPlayers[fhCount] = (FieldHockeyPlayer) s; ##B
14 fhCount++;
15 }
16 }
17 //code omitted - see Listing 16.1
18 }
19 }

```

**#A** This is an example of upcasting, we are casting a `FieldHockeyPlayer` to a `Student` object reference.

**#B** Since we included the check to make sure this object is a `FieldHockeyPlayer`, we can use this to downcast from `Student`

One important note when using upcasting and downcasting is the order of precedence of the dot operator. In my example, I am casting the `Student` object `s` to a `FieldHockeyPlayer` object before adding it to the `fhPlayers` array. If I needed to access any of the instance data or methods from the `FieldHockeyPlayer` class, I can use this same logic. I can cast the `Student` to a `FieldHockeyPlayer`, but I must enclose the casting portion with parentheses so that it is done first. Table 16.2 shows valid/invalid access to the `FieldHockeyPlayer` class methods given the `Student` reference variable `s`. These statements would be defined in the main method from Listing 16.6.

**Table 16.2** Examples of valid/invalid statements when using downcasting

| Statement                                                                                      | Valid/Invalid | Description                                                                                                                                                                                                 |
|------------------------------------------------------------------------------------------------|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>Student s = new FieldHockeyPlayer(...); (FieldHockeyPlayer)s.setGoalsScored(4);</pre>     | Invalid       | This statement causes an error since the compiler tries to execute the dot operator first. But the <code>Student</code> object <code>s</code> does not have a method called <code>setGoalsScored()</code> . |
| <pre>Student s = new FieldHockeyPlayer(...); ((FieldHockeyPlayer)s).setGoalsScored(4);</pre>   | Valid         | This is a valid statement because the student object is first casted to a <code>FieldHockeyPlayer</code> , then it looks for the <code>setGoalsScored()</code> method.                                      |
| <pre>Student s = new FieldHockeyPlayer(...); ((FieldHockeyPlayer)s).position = "goalie";</pre> | Invalid       | In this case, the parentheses correctly cast the object <code>s</code> to a <code>FieldHockeyPlayer</code> , but the <code>position</code> data field is defined as private, so this is invalid.            |
| <pre>Student s = new Athlete(...); (Athlete)s.setEligibility(true);</pre>                      | Invalid       | In this statement, the compiler tries to find the <code>setEligibility</code> method in the <code>Student</code> class, but it is not there.                                                                |
| <pre>Student s = new Athlete(...); ((Athlete)s).setEligibility(true);</pre>                    | Valid         | Since the object is first cast to an <code>Athlete</code> , this statement is now valid.                                                                                                                    |
| <pre>Student s = new Student(...); Athlete a = (Athlete)(new Student(...));</pre>              | Invalid       | Causes a <code>ClassCastException</code> error at runtime because we can't cast a subclass as its superclass                                                                                                |
| <pre>Student s = new Athlete(...); Athlete a = (Athlete)s;</pre>                               | Valid         | This is a valid downcasting example since I know <code>s</code> is an <code>Athlete</code>                                                                                                                  |

**Quick Check 16-3:**

Given this statement, answer the questions:

```
Athlete a = new SoccerPlayer(...);
Student s = new Student(...);
```

1. Which statement is a valid example of downcasting:

- a. `SoccerPlayer sp = (SoccerPlayer)a;`
- b. `Student s = a;`
- c. `Athlete a = (Athlete)(s);`

2. Which statement is a valid example of upcasting:

- a. `SoccerPlayer sp = (SoccerPlayer)a;`
- b. `Student s = a;`
- c. `Athlete a = (Athlete)(s);`

Given the classes for Student, Athlete, and TennisPlayer, and the code below to answer the following questions:

```
TennisPlayer tp1 = new TennisPlayer("Abhiram", "Nirman", 17, 11, 4.0,
 true, "L", true);
Athlete tp2 = new TennisPlayer("Bethan", "Palmer", 15, 10,
 3.5, true, "S", false);
Student tp3 = new TennisPlayer("Yung-jin", "Ryoo", 14, 9,
 3.75, true, "M", false);
Student s1 = new Student("Zahraa", "Khalil", 14, 9, 3.95);
```

1. Which statement does NOT use the toString method in the TennisPlayer class first:

- a. `tp1.toString();`
- b. `s1.toString();`
- c. `tp2.toString();`
- d. `tp3.toString();`

2. Which statement(s) can access the getStatus() method of the Athlete class:

- a. `tp1.getStatus();`
- b. `s1.getStatus();`
- c. `tp2.getStatus();`
- d. `tp3.getStatus();`

3. Which class toString method is called first with this code snippet:

```
((Athlete)tp3).toString();
```

- a. Athlete
- b. TennisPlayer



- c. `Student`
- d. none, it gives you an error message

## 16.4 How the JVM Chooses the Correct Method to Execute

In this lesson, we have seen examples of polymorphism in action. Runtime polymorphism occurs when a subclass overrides the same method signature as a method in the superclass. This can actually cause a daisy chain effect, where each time a subclass extends a superclass it can override the method. In figure 16.3, the main method creates four new objects using `Class1`. Each object variable is referencing either the superclass (`Class1`) or the subclasses `Class2`, `Class3`, and `Class4`. Each class has a `method1` that takes one integer value. The superclass has a second method, `method2` that also takes one integer value.

At runtime, the JVM starts by looking for `method1` in the corresponding class. The last statement defines an object `c4` that references a `Class4` object. When the JVM invokes `method2` on this object, it starts by looking for an overridden method in `Class4`, but there is not method. So, it checks the superclass, `Class3`. It keeps going until it finds the correct method signature in `Class1`.

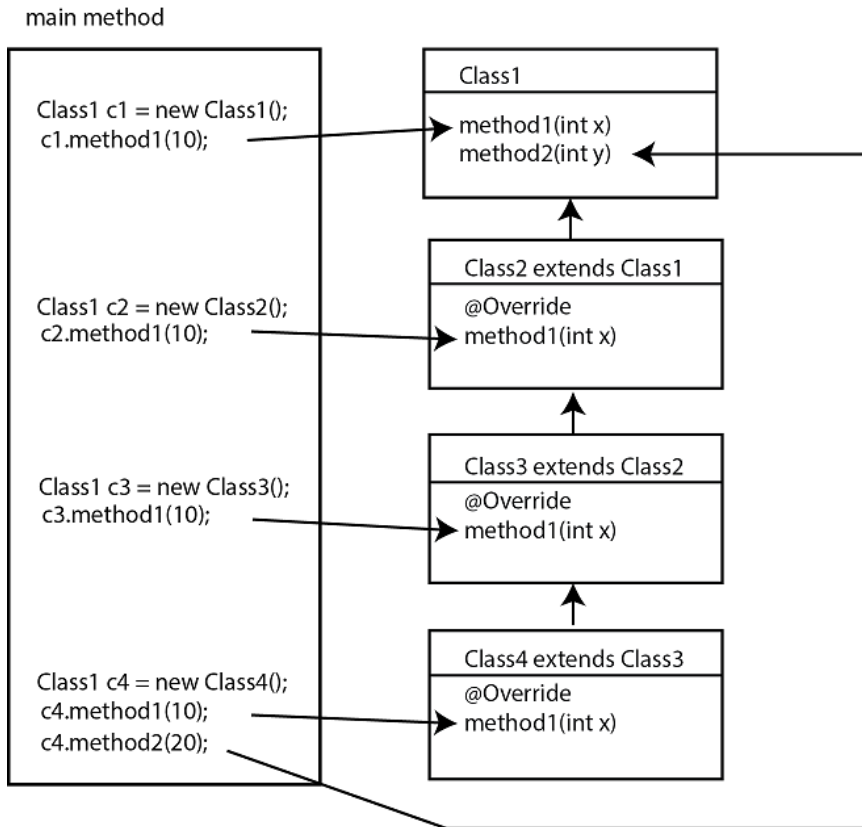


Figure 16.5 Diagram of runtime polymorphism

## 16.5 Summary

In this lesson, you learned:

- How to define objects that benefit from polymorphism
- The difference between upcasting and downcasting objects
- How the virtual machine chooses the correct method when a subclass overrides a superclass method

I realize this was a long lesson with a lot of information. This lesson reviewed how to create objects that are derived from the same class hierarchy but behave differently, this is called polymorphism in Java. And finally, we learned how to determine which method is invoked when multiple methods exist with the same method signature. In the next lesson, I introduce another topic related to comparing objects.

**Try this:**

Figure 16.4 shows the class hierarchy of an `Electronics` class, plus a new class `CellPhone`. The `CellPhone` class represents another type of `Electronics` and it should include an overridden `print` method. For this activity, add a `CellPhone` class to the code from listing 16.7 and then create the following objects in the main method that invoke the `print` method in the `CellPhone` class:

- a. Create an `Electronics` object that is set to a `CellPhone` object
- b. Create a `CellPhone` object

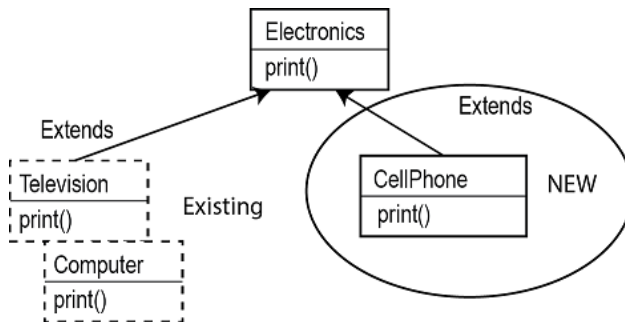


Figure 16.6 shows the hierarchy of the `Electronics` class with a new `CellPhone` class added

#### Listing 16.9 Starting code for the Try This activity

```

1 public class Lesson16_Examples {
2 public static void main(String[] args) { // #A
3 Electronics e = new Electronics("Samsung", "4k");
4 e.print();
5 e.setMake("Samsung");
6 }
7 }
8 class Electronics { // #B
9 public String make, model;
10 public Electronics(String make, String model) {
11 this.make = make;
12 this.model = model; }
13 public void setMake(String make) {
14 this.make = make; }
15 public void print() {
16 System.out.println("I'm an electronics object");
17 } }
18 class Television extends Electronics { // #C
19 public Television(String make, String model) {
20 super(make, model);
21 }
22 public void print() {
23 System.out.println("I'm a television");
24 }
25 }

```

```

26 class Computer extends Electronics { //#D
27 public Computer(String make, String model) {
28 super(make, model);
29 }
30 public void print() {
31 System.out.println("I'm a computer");
32 }
33 }

```

#A The main method creates an Electronics object, calls the print and setMake methods

#B The electronics class is the superclass

#C The Television class extends the Electronics class and overrides the print method

#D The Computer class also extends the Electronics class and overrides the print method

### Quick Check 16-1 Solution:

1. Using Figure 16.1, which statement is not valid:
  - a. An Athlete is a Student
  - b. A TennisPlayer is a Student
  - c. A TennisPlayer is an Athlete
  - d. A Student is an Athlete (**not every student is an athlete**)

### Quick Check 16-2 Solution:

1. Using Figure 16.3, which statement is not valid:
  - a. `Computer laptop = new Electronics();`
  - b. `GamingSystem gs = new GamingSystem();`
  - c. `Electronics tv = new Television();`
  - d. `Electronics nintendoDS = new GamingSystem();`
2. If we add a new class that extends Computer called Laptop, which of the following is valid:
  - a. `Laptop laptop = new Computer();`
  - b. `Computer laptop = new Laptop();`
  - c. `Laptop laptop = new Electronics();`
  - d. `Computer laptop = new Television();`

### Quick Check 16-3:

**Given this statement, answer the questions:**

```
Athlete a = new SoccerPlayer(...);
```

1. Which statement is a valid example of downcasting:
  - a. `SoccerPlayer sp = (SoccerPlayer)a;`
  - b. `Student s = a;`
  - c. `Athlete a = (Athlete)s;`

2. Which statement is a valid example of upcasting:

- a. `SoccerPlayer sp = (SoccerPlayer)a;`
- b. `Student s = a;`
- c. `Athlete a = (Athlete)(s);`

**NOTE:** It can be difficult to differentiate between upcasting and downcasting. In this activity, the answer to Question 1 shows code that changes an Athlete object reference and casts it as a SoccerPlayer reference. So, it is going from the superclass to a subclass, hence downcasting. Question 2 does the opposite, it takes a subclass object and casts it as a superclass implicitly in the assignment statement.

Given the classes for Student, Athlete, and TennisPlayer, and the code below to answer the following questions:

```
TennisPlayer tp1 = new TennisPlayer("Abhiram", "Nirman", 17, 11, 4.0,
 true, "L", true);
Athlete tp2 = new TennisPlayer("Bethan", "Palmer", 15, 10,
 3.5, true, "S", false);
Student tp3 = new TennisPlayer("Yung-jin", "Ryoo", 14, 9,
 3.75, true, "M", false);
Student s1 = new Student("Zahraa", "Khalil", 14, 9, 3.95);
```

1. Which statement does NOT use the toString method in the TennisPlayer class first:

- a. `tp1.toString();`
- b. `s1.toString();`
- c. `tp2.toString();`
- d. `tp3.toString();`

2. Which statement(s) can access the getStatus() method of the Athlete class:

- a. `tp1.getStatus();`
- b. `s1.getStatus();`
- c. `tp2.getStatus();`
- d. `tp3.getStatus();`

3. Which class toString method is called first with this code snippet:

```
((Athlete)tp3).toString();
```

- a. Athlete
- b. **TennisPlayer**
- c. Student
- d. none, it gives you an error message

**Solution to the Try This activity:****Listing 16.7 Solution to Try This Activity**

```

1 public class Lesson16_Examples {
2 public static void main(String[] args) {
3 Electronics e = new CellPhone("Samsung", "9 Plus", "Smart Phone"); //#A
4 CellPhone cell = new CellPhone("LG", "3", "Flip phone"); //#B
5 e.print(); //#C
6 cell.print(); //#D
7 System.out.println(cell.style); //#E
8 } }
9 class Electronics {
10 public String make, model;
11 public Electronics(String make, String model) {
12 this.make = make;
13 this.model = model; }
14 public void setMake(String make) {
15 this.make = make; }
16 public void print() {
17 System.out.println("I'm an electronics object");
18 } }
19 class CellPhone extends Electronics {
20 public String style;
21 public CellPhone(String make, String model, String style){
22 super(make, model);
23 this.style = style; }
24 public void print() {
25 System.out.println("I'm a cellphone object");
26 }
27 }
28 //code for other classes omitted, can be found in listing 16.7

```

**#A** Creates an Electronics object set to a new CellPhone instance (must include three arguments)

**#B** Creates a CellPhone object

**#C** Calls the print method in the CellPhone class (because it overrides the Electronics class)

**#D** Calls the print method in the CellPhone class

**#E** Prints the style value for the CellPhone object

# 17

## Comparing Objects

**After reading lesson 17, you will be able to:**

- Understand the difference between comparing objects vs. primitive data types
- Compare objects for equality
- Use the Comparable interface and override the compareTo method to compare objects

When comparing primitive data types such as numbers, Java uses the mathematical symbols that we are all familiar with: `>`, `>=`, `==`, `<`, `<=`. Java also includes an exclamation point to negate any comparison. But these symbols alone do not help us compare objects. Instead, we need to define methods that provide comparisons such as equals, less than, and greater than. In this lesson, you will learn how to compare objects.

### Consider This

Imagine a neighborhood where all the houses were built by the same contractor. She made each house the same with the two bedrooms, two bathrooms, a living room, a kitchen, a laundry room, and a one car garage. The owners of the houses might have changed the type of roof (metal vs. shingle) or the exterior color of the house. Each house in the neighborhood has a different house number. So, if your address is 123 Main Street, the house next door might be 124 Main Street.

A realtor wanted to have a report that compared these houses. If they compare the address location, it would not be the same. But if they wanted to know if they have the same layout, then that would be true since all the houses were built the same. So, it is important to understand if we are comparing two houses based on their characteristics or their location.

The same is true about objects in Java. A reference variable for an object in Java contains the memory address of that object. So, even if we create two House objects and they have the exact same information, the address location of each object would be different. To compare these House objects, we need to identify what information should be used in the comparison. For example, we can check to see if they have the same number of rooms and the same roof material.

There are times when we also want to know if two object reference variables contain the same memory address. This comparison is done by comparing the reference variables. This is similar to checking the street address of two houses to see if they are actually the same house.

## 17.1 Comparing Objects vs. Primitive Data Types

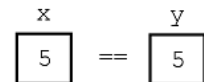
Remember, primitive data includes numbers (int, double, float, short, long, byte), and characters (char). Sometimes we include Strings, but they are actually a special case. When comparing these types of data, we can use the comparison operators: ==, >, >=, <, <=. These operators directly compare the values and determine if they are equal, greater than or less than. All of these comparisons are actually done numerically, since every character can be represented by a numeric value.

Now let's talk about comparing objects. When we create an object variable, the variable stores a reference value to the location of the object data in memory. When we use the keyword `new` to create a new instance of a class, the memory location is stored in the variable. Even if these two objects have the exact same values, the reference values will be different. So, if we compare these two objects to test for equality using the symbol ==, it will return false. Figures 17.1 and 17.2 show the difference between comparing two primitive data types and two objects.

```
public static void main(String[] args) {
 int x = 5, y = 5;
 if(x == y) ← this statement is true
 System.out.println("x is equal to y");
}
```

sample output

x is equal to y



this model is the same for all primitive data types, (int, double, float, short, byte, long, char, boolean)

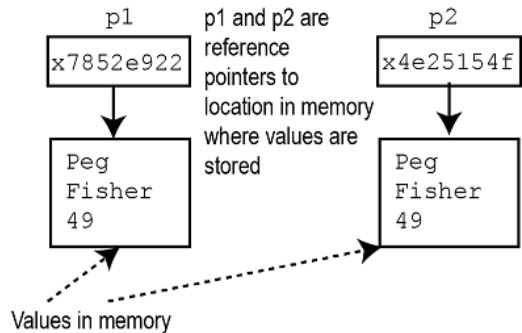
Figure 17.1 shows what happens when comparing two primitive data values



```

Person p1 = new Person("Peg", "Fisher", 49);
System.out.println(p1);
Person p2 = new Person("Peg", "Fisher", 49);
System.out.println(p2);
if(p1 == p2) ← this statement is false
 System.out.println("p1 is equal to p2");

```



#### sample output

```

lesson17_examples.Person@7852e922
lesson17_examples.Person@4e25154f

```

In the sample output, the values of p1 and p2 represent the reference address of the object data for each object. As you can see in the sample output the values are not the same

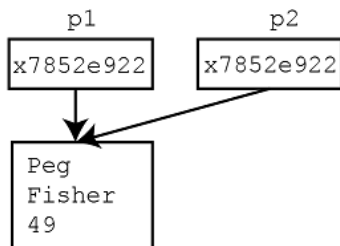
Figure 17.2 shows what happens when comparing two objects

Can you think of an instance where p1 might be equal to p2? If we are interested in whether or not both values have the same reference value, then they are equal. In this case, the objects are considered aliases of each other. In this example, I can accomplish this by having p2 point to p1:

```

Person p1 = new Person("Peg", "Fisher", 49);
Person p2 = p1;
if(p1 == p2) System.out.println("p1 is equal to p2");
System.out.println(p1);
System.out.println(p2);

```



#### sample output

```

p1 is equal to p2
lesson17_examples.Person@7852e922
lesson17_examples.Person@7852e922

```

Figure 17.3 assigns p2 to the same reference value as p1

Now, p1 and p2 have the same reference value and point to the EXACT same object in memory. Therefore, when we compare them using the == sign, the statement is considered true. In the next section, I will show you how to add a method to compare two objects for equality.

### Quick Check 17-1:

#### Listing 17.1 contains code for a Person class and a main method

```

1 package lesson17_examples;
2 public class Lesson17_Examples {
3 public static void main(String[] args) {
4 int x = 35;
5 int y = 5;
6 if(x == y) //#A
7 System.out.println("x is equal to y");
8
9 Person p1 = new Person("Cy", "Young", 35);
10 Person p2 = new Person("cy", "young", 35);
11 if(p1 == p2) //#B
12 System.out.println("p1 is equal to p2");
13 if(p1.age == p2.age) System.out.println("p1 age is equal to p2 age"); //#C
14 if(x == p1.age) System.out.println("x is equal to p1 age"); //#D
15 }
16 }
17 class Person {
18 String fname, lname;
19 int age;
20 public Person(String fname, String lname, int age) {
21 this.fname = fname;
22 this.lname = lname;
23 this.age = age;
24 }
25 }

```

#A compares two primitive data types

#B compares two object reference variables

#C compares the age of person p1 to the age of person p2

#D compare the value of x to the age of person p1

1 Using Listing 17.1 what is printed to the console

- a. p1 is equal to p2
- b. p1 is equal to p2  
p1 age is equal to p2 age  
x is equal to p1 age
- c. p1 age is equal to p2 age  
x is equal to p1 age
- d. x is equal to p1 age

2 True/False: p1.fname == p2.fname

3 True/False: p1.age == p2.age

## 17.2 Compare Objects for Equality

As we have seen in the previous section, if we use the `==` sign to compare two objects, it only compares their reference values. There are times when we want to check and see if all the values from one object match the values from another object. To accomplish this, we can add a new method to our `Person` class. This new method is added using a generic object as the argument. If the argument is of type `Person`, when we can check each field to see if they match. If it does, then the method returns `true`, otherwise it returns `false`. Listing 17.2 is an example of a `Person` class with an `equals` method used to test for equality:

### Listing 17.2 contains code for a `Person` class with an extra method for comparing `Person` objects

```

1 package lesson17_examples;
2 public class Lesson17_Examples {
3 public static void main(String[] args) {
4 Person p1 = new Person("Sheila", "Gelin", 35);
5 Person p2 = new Person("Sheila", "Gelin", 35);
6 Person p3 = new Person("Ray", "Villalobos", 25);
7 if(p1.equals(p2)) System.out.println("p1 is equal to p2"); // #A
8 if(p1.equals(p3)) System.out.println("p1 is equal to p3"); // #A
9 }
10 }
11 class Person {
12 String fname, lname;
13 int age;
14 public Person(String fname, String lname, int age) {
15 this.fname = fname;
16 this.lname = lname;
17 this.age = age;
18 }
19 public boolean equals(Object obj) { // #B
20 if (obj instanceof Person) {
21 if (this.fname == ((Person) obj).fname
22 && this.lname == ((Person) obj).lname
23 && this.age == ((Person) obj).age) {
24 return true; }
25 }
26 return false;
27 }
28 }

```

#A This statement uses the new `equals` method defined in the `Person` class to test for equality

#B This method checks to see if the first name, last name and age are the same

In the new `equals` method, each field from the `Person` class is compared. If they are all equal, the method returns `true`. If the object data is not the same or if it is not an instance of the `Person` class, it returns `false`. Notice that the `equals` method is invoked on lines 7 and 8 in Listing 17.2. The method uses the object on the left-hand side (lhs) as the calling object, in this case `p1` is the calling object for both statements. In the parentheses, the program passes an object as the argument to the method. The first statement uses `p2`, the second statement

uses `p3`. In the `equals` method, the values of the calling object are referred to by the keyword `this`. The objects `p2` and `p3` are passed to the argument variable `obj` which is defined as an `Object`. Remember, every class inherits from the `Object` class, so we can use that to define an `Object` variable and have it reference a `Person` object. But, in order to compare the fields from our calling object to the information in our argument object, we must cast it as a `Person` object. Finally, notice the use of the `instanceof` keyword to check if the argument is a `Person` object.

Figures 17.4 and 17.5 are diagrams depicting how the objects are passed to the `equals` method with annotation to help explain how the variables in the method refer to different objects.

Start by creating three `Person` objects

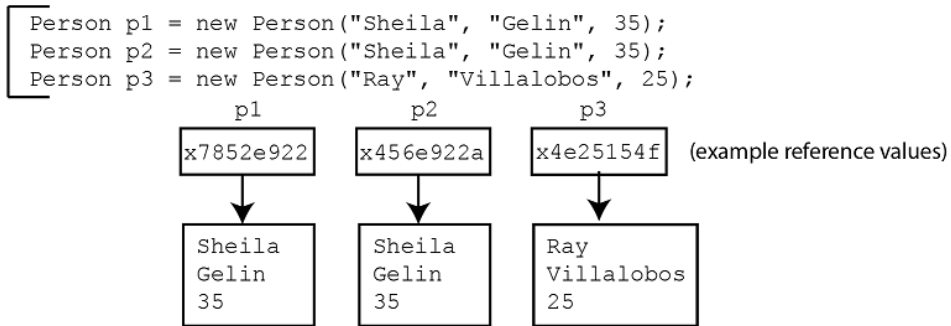


Figure 17.4 shows what happens when three `Person` objects are created

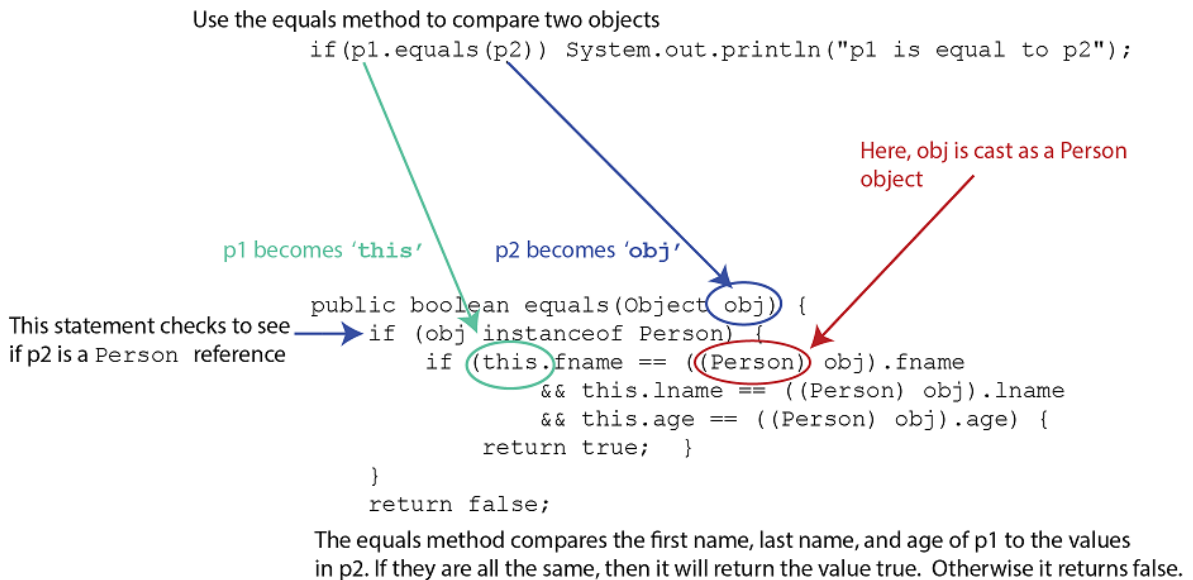


Figure 17.5 is a diagram showing how the objects are passed to the equals method for comparison

**NOTE:** it is also possible to use the .equals method from the String class to compare the first name and last name instead of the == operator, this code produces the same results as above:

```
if (obj instanceof Person) {
 if (this.fname.equals(((Person) obj).fname)
 && this.lname.equals(((Person) obj).lname)
 && this.age == ((Person) obj).age) {
 return true; }
}
return false;
```

By adding the equals method to the Person class, the programmer can define the criteria for comparison based on the business requirements. For example, if the business requirements want to compare two Person objects and consider them equal if they have the same first name and last name, regardless of age, the code could be changed to only test those two characteristics of the objects. Listing 17.3 shows a revised version that only checks for first and last name to be the same.

#### Listing 17.3 revised listing to only compare first and last names

```
1 package lesson17_examples;
2 public class Lesson17_Examples {
3 public static void main(String[] args) {
4 Person p1 = new Person("Sheila", "Gelin", 35);
5 Person p2 = new Person("Sheila", "Gelin", 35);
```

```

6 Person p3 = new Person("Ray", "Villalobos", 25);
7 if(p1.equals(p2)) System.out.println("p1 is equal to p2");
8 if(p1.equals(p3)) System.out.println("p1 is equal to p3");
9 }
10 }
11 class Person {
12 String fname, lname;
13 int age;
14 public Person(String fname, String lname, int age) {
15 this.fname = fname;
16 this.lname = lname;
17 this.age = age;
18 }
19 public boolean equals(Object obj) { // #A
20 if (obj instanceof Person) {
21 if (this.fname == ((Person) obj).fname
22 && this.lname == ((Person) obj).lname) {
23 return true; }
24 }
25 return false;
26 }
27 }

```

#A The equals method has been updated to only check for first name and last name

### Quick Check 17-2:

1. What statement below could replace lines 21-22 in Listing 17.3 to only check for last name:

- a. `if (this.fname == ((Person) obj).fname  
|| this.lname == ((Person) obj).lname) {`
- b. `if (this.lname == ((Person) obj).lname) {`
- c. `if (this.lname == obj.lname) {`
- d. **There is no way to only check for last name**

2. If we add a new person object after line 6 in Listing 17.3,

```
Person p4 = p3;
```

which statement(s) are true:

- a. `if(p3 == p4)`
- b. `if(p3.equals(p4))`
- c. `if(p3.fname == p4.fname)`
- d. `if(p3.lname == p4.lname)`

## 17.3 Use the Comparable interface to compare objects

In the previous section, I demonstrated one way to check if two objects are equal. There are times when we not only need to check if they are equal, we might also need to know which

comes first based on our business requirements. This type of comparison is usually needed to help sort a list of objects.

In my example, the business requirements might require that the application has the ability to sort the Person objects by last name. Lucky for us, Java provides a Comparable interface that can be used to define the process for comparing two objects. We start by implementing the Comparable interface and then provide an overridden method for the abstract method, `compareTo(Object o)`. Code listing 17.4 shows an example of how to use the Comparable interface. To save space, I have only added two Person objects and sorted them by last name. I've also added an overridden `toString` method to return the person name.

#### Listing 17.4 Using the Comparable interface to compare objects

```

1 package lesson17_examples;
2 public class Lesson17_Examples {
3 public static void main(String[] args) {
4 Person p1 = new Person("Hannah", "Wise", 26);
5 Person p2 = new Person("Courtney", "Johnson", 31);
6 if (p1.compareTo(p2) < 0) { ##A
7 System.out.println(p1.toString() + p2.toString());
8 } else {
9 System.out.println(p2.toString() + p1.toString());
10 }
11 }
12 }
13 class Person implements Comparable { ##B
14 private String fname, lname;
15 private int age;
16 public Person(String fname, String lname, int age) {
17 this.fname = fname;
18 this.lname = lname;
19 this.age = age;
20 }
21 @Override
22 public int compareTo(Object obj) { ##C
23 if (obj instanceof Person) {
24 if (this.lname.compareToIgnoreCase(((Person) obj).lname) < 0) { ##D
25 return -1;
26 } else if (this.lname.compareToIgnoreCase(((Person) obj).lname) == 0) { ##E
27 return 0;
28 } else { ##F
29 return 1;
30 }
31 }
32 return 0;
33 }
34 @Override
35 public String toString() {
36 return lname + ", " + fname + "\n";
37 }
38 }

```

**#A Use the `compareTo` method to sort the last name**

**#B Add the interface to the Person class by implementing Comparable**

**#C** When implementing Comparable, the program **MUST** provide an overridden implementation of the compareTo() method

**#D** Compare the last name of each object, if the calling object is first alphabetically, return -1

**#E** If the last name is exactly the same regardless of capitalization, return 0

**#F** If the last name of the calling object comes after the last name of the argument object, return 1

The compareTo(Object o) method is defined as an abstract method in the Comparable interface. So, when we implement that interface, we are **required** to provide an implementation of this method. From the method signature we can see that it returns an integer value. That value is then used by the calling program to determine which object came first based on the evaluation criteria. The evaluation criteria for this example was the spelling of the last name (I chose to ignore whether the letters were upper-case or lower-case). The overridden method compares the last name of the calling object to the last name of the object in the argument list. It compares each character in the name alphabetically. Using this information, the main method prints the person names in alphabetic order based on last name only.

### **Quick Check 17-3:**

**Using the code in listing 17.4 and the additional statements below, answer the following questions:**

```
Person p3 = new Person("Kaitlyn", "Wise", 29);
Person p4 = new Person("Courtney", "Jones", 19);
```

1. What is printed if I use the compareTo method with p1 and p3:

- a. Wise, Kaitlyn  
Wise, Hannah
- b. Wise, Hannah  
Wise, Kaitlyn
- c. Wise, Hannah  
Wise, Hannah
- d. Wise, Kaitlyn  
Wise, Kaitlyn

2. What is printed if I use the compareTo method with p2 and p4:

- a. Wise, Kaitlyn  
Jones, Courtney
- b. Johnson, Courtney  
Wise, Kaitlyn
- c. Jones, Courtney  
Johnson, Courtney
- d. Johnson, Courtney  
Jones, Courtney



3. **What is wrong with this code snippet:**

```
if(p2.lname == p3.lname) System.out.println("p2 has the same last name as p3");
```

- a. **nothing, it prints the message**
- a. **lname is defined as private, so it cannot be used in the main method**
- b. **the code must use the .compareTo(Object o) method**
- c. **the code must use the .equals(String str) method**

## 17.4 Summary

In this lesson, you learned:

- that primitive data types can be compared using `<`, `<=`, `>`, `>=`, or `==` but the same is not true for objects
- how to add a new method to compare two objects for equality
- to implement the `Comparable` interface to compare two objects based on business requirements

This lesson reviewed how to compare objects in Java. Most classes are defined with one or many attributes. In order to compare two objects, the application must add logic to identify what attributes to compare and the criteria for comparing them. In the example, the `Person` class defined three attributes: first name, last name, and age. In order to compare two objects, it is necessary to identify what attributes are included in the comparison. The business requirements might only want to know if all person objects are the same age. But another requirement might be to sort all the person objects by age from oldest to youngest. So, in this lesson we learned how to check objects for equality and how to add logic for comparing two objects. In the next lesson, I will be combining Lessons 14-17 in a cumulative capstone project.

### **Try this:**

A bank account might have an account number, the customer first and last name, and a beginning balance. If we want to compare two bank account objects, we need to determine which values to compare. For this activity, start with a `BankAccount` class that implements `Comparable`. Add the four instance data fields and then provide an `equals` method that determines if the accounts are duplicates. This is needed since the list of accounts might have accidentally assigned the same information to two account objects.

### **Quick Check 17-1 Solution:**

#### **Listing 17.1 contains code for a Person class and a main method**

```
1 package lesson17_examples;
2 public class Lesson17_Examples {
3 public static void main(String[] args) {
4 int x = 35;
```

```

5 int y = 5;
6 if(x == y)
7 System.out.println("x is equal to y");
8
9 Person p1 = new Person("Cy", "Young", 35);
10 Person p2 = new Person("cy", "young", 35);
11 if(p1 == p2)
12 System.out.println("p1 is equal to p2");
13 if(p1.age == p2.age) System.out.println("p1 age is equal to p2 age");
14 if(x == p1.age) System.out.println("x is equal to p1 age");
15 }
16 }
17 class Person {
18 String fname, lname;
19 int age;
20 public Person(String fname, String lname, int age) {
21 this.fname = fname;
22 this.lname = lname;
23 this.age = age;
24 }
25 }

```

1. Using Listing 17.1 what is printed to the console

- a. p1 is equal to p2
- b. p1 is equal to p2  
p1 age is equal to p2 age  
x is equal to p1 age
- c. p1 age is equal to p2 age  
x is equal to p1 age
- d. x is equal to p1 age

2. True/False: p1.fname == p2.fname **FALSE**

3. True/False: p1.age == p2.age **TRUE**

### Quick Check 17-2 Solution:

1. What statement below could replace lines 21-22 in Listing 17.3 to only check for last name:

- a. `if (this.fname == ((Person) obj).fname  
|| this.lname == ((Person) obj).lname) {`
- b. `if (this.lname == ((Person) obj).lname) {`
- c. `if (this.lname == obj.lname) {`
- d. **There is no way to only check for last name**

2. If we add a new person object after line 6 in Listing 17.3,

```
Person p4 = p3;
```

which statement(s) are true: **(all of them)**

- a. `if(p3 == p4)`
- b. `if(p3.equals(p4))`

- c. `if(p3.fname == p4.fname)`
- d. `if(p3.lname == p4.lname)`

**Quick Check 17-3 Solution:**

**Using the code in listing 17.4 and the additional statements below, answer the following questions:**

```
Person p3 = new Person("Kaitlyn", "Wise", 29);
Person p4 = new Person("Courtney", "Jones", 19);
```

1. What is printed if I use the `compareTo` method with `p1` and `p3`:

- a. Wise, Kaitlyn  
Wise, Hannah
- b. **Wise, Hannah**  
**Wise, Kaitlyn**
- c. Wise, Hannah  
Wise, Hannah
- d. Wise, Kaitlyn  
Wise, Kaitlyn

2. What is printed if I use the `compareTo` method with `p2` and `p4`:

- a. Wise, Kaitlyn  
Jones, Courtney
- b. Johnson, Courtney  
Wise, Kaitlyn
- c. Jones, Courtney  
Johnson, Courtney
- d. **Johnson, Courtney**  
**Jones, Courtney**

3. What is wrong with this code snippet:

```
if(p2.lname == p3.lname) System.out.println("p2 has the same last name as p3");
```

- a. nothing, it prints the message
- b. **lname is defined as private, so it cannot be used in the main method**
- c. the code must use the `.compareTo(Object o)` method
- d. the code must use the `.equals(String str)` method

**Solution to the Try This activity:**

**Listing 17.3 Solution to Try This Activity**

```
1 package lesson17;
2 public class Lesson17 {
```

```

3 public static void main(String[] args) {
4 BankAccount ba1 = new BankAccount("Ande", "Withers", 1234, 1000.40); #A
5 BankAccount ba2 = new BankAccount("Ande", "Withers", 1234, 1000.40); #A
6 BankAccount ba3 = new BankAccount("Ande", "Withers", 1234, 100.40); #A
7 if(ba1.equals(ba2)) System.out.println("ba1 is equal to ba2"); #B
8 if(ba1.equals(ba3)) System.out.println("ba1 is equal to ba3"); #C
9 }
10 }
11 class BankAccount {
12 private String fname, lname;
13 private double balance;
14 private int accountNumber;
15 public BankAccount(String fname, String lname, int accountNumber,
16 double balance) {
17 this.fname = fname;
18 this.lname = lname;
19 this.accountNumber = accountNumber;
20 this.balance = balance;
21 }
22 public boolean equals(Object obj) { //D
23 BankAccount ba = null; //E
24 if(obj instanceof BankAccount) { //F
25 ba = (BankAccount)obj; //G
26 }
27 if(ba != null) { //H
28 if(this.fname == ba.fname && this.lname == ba.lname && //I
29 this.accountNumber == ba.accountNumber &&
30 this.balance == ba.balance)
31 return true; //J
32 }
33 return false; //K
34 }
35 }

```

**#A** Creates three bank account objects where ba1 = ba2, but ba3 is slightly different  
**#B** Test the equals method with two objects that are the same  
**#C** Test the equals method with two objects that have a slight difference (this message should not print)  
**#D** Create an equals method that takes in an object and returns a boolean  
**#E** Create a BankAccount object and set it to null  
**#F** Test if the object in the argument list is of type BankAccount  
**#G** If the object is an instanceof BankAccount, then assign it to the variable ba  
**#H** Check if the variable ba is not null (which means the argument value was a valid BankAccount object)  
**#I** Compare all four values of calling object to ba (which was passed as an argument to the method)  
**#J** If all fields are the same, return true  
**#K** If all fields are not the same, return false

# 18

## Capstone

### **After reading lesson 18, you will be able to:**

- Incorporate method overloading and method overriding in an application
- Understand how to use polymorphism in an application
- Implement the comparable interface to compare objects in an application

The capstone for this unit concentrates on the learning objectives from lessons 13-17 but also includes everything we have learned so far in this book. The lesson starts with a set of business requirements. Based on these requirements, I will walk through the steps to complete an application. The business problem we are going to address is providing a report of upcoming projects for a general contracting business. A general contractor is hired for various type of projects from building a house to minor repairs. For this application, the contractor wants to keep track of the employees working on a project, the project customer's name and address, and the start and end date for the project.

The application must be able to:

1. Create a project with a start date, end date, customer name, address of the project, description, estimated cost, overhead percentage for the contractor, and a list of workers
2. Print out the list of projects including the project address location
3. Compare two projects and determine which project must be done first based on the start date

This is a small company that hires the following types of workers:

- Electrician
- Plumber
- Carpenter

The workers are paid hourly, but they also each have additional expenses. It is also important to understand that the general contractor might not need all of these workers for every project. For example, if the contractor is building a new house, she needs all three types of workers. But if she is only adding an extra room onto an existing house, she might only need an electrician and a carpenter.

For this lesson, the application is designed with a minimal amount of information about each project, but it is used to demonstrate the concepts from this unit. It also provides the framework for a larger, more comprehensive application such as adding inventory tracking for the general contractor. The next section reviews the classes need for this application.

## 18.1 Creating Classes

Since the contractor might have one or many projects going on at the same time, the program is designed to allow multiple projects to be created. Each project might have one or more workers assigned to the project. Examples of projects include building a house, adding an addition, renovating a bathroom, or even just adding outdoor motion lights. Based on the business requirements for this application, I have determined we need the following classes:

- Project class represents the information needed for the project
- Address class is used any time an address is needed, for example, the location of the project or the address of the workers
- Worker class contains general information about all workers
- Electrician, Plumber, Carpenter classes extend the Worker class

The electrician, plumber and carpenter are types of workers, so each of these classes will extend the worker class. Figure 18.1 shows the initial UML diagram for these classes. More instance variables and additional methods might be needed later, but this is a good starting point.

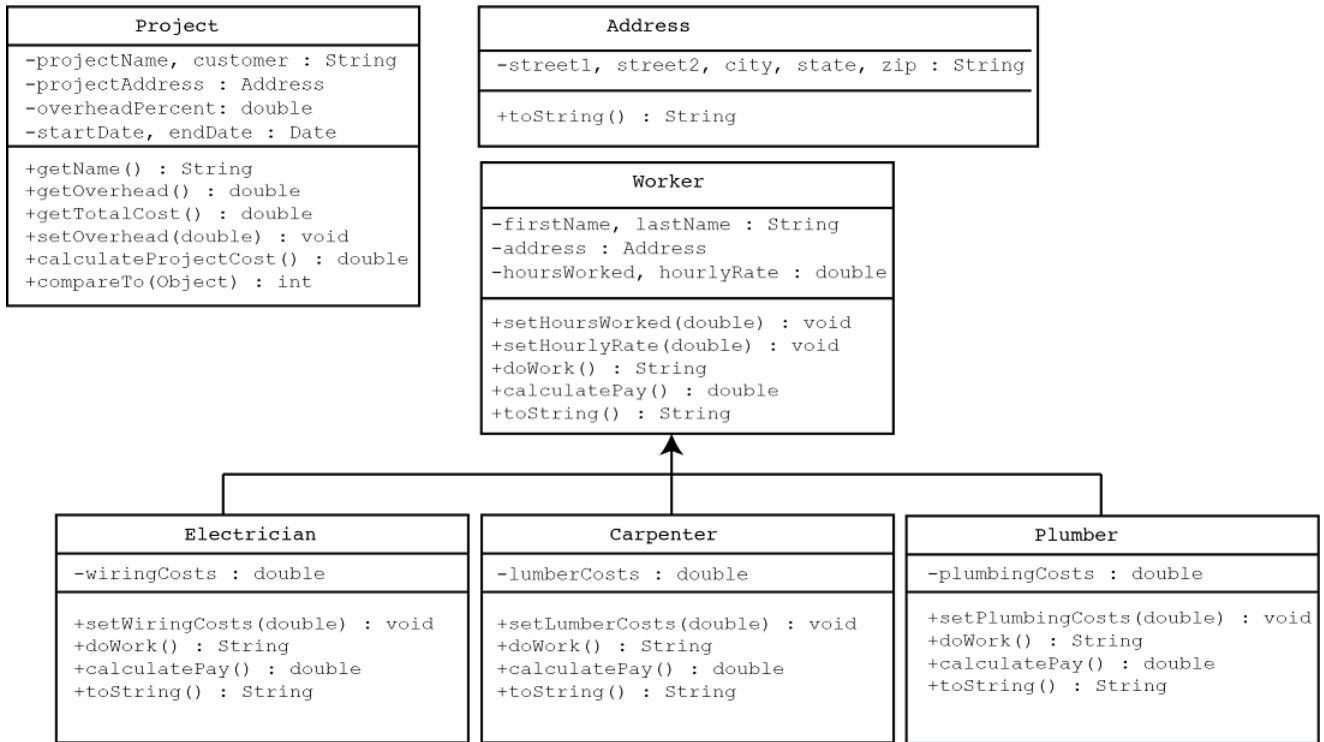


Figure 18.1 contains a diagram of the initial UML diagrams for each class

Using this approach to create the classes for this project, I can create a list of workers who are associated with a specific project. This design creates a list of workers which can be a regular Worker, an Electrician, a Carpenter, or a Plumber. Notice that each subclass of the Worker class overrides these methods: `doWork`, `calculatePay` and `toString`. Using polymorphism I can invoke the overridden methods for each type of worker when calculating their total project cost which is made up of wages and expenses. Polymorphism is also used when each subclass overrides the `toString` method and the JVM decides which specific version of the `toString` method to invoke.

One of the requirements for this project is to have the ability to compare two projects and identify which projects should be started first. To provide this comparison, the Project class implements `Comparable` and then overrides the `compareTo` method.

### 18.1.1 The Project class

The project class in this application will do most of the heavy lifting. For this class, I have decided to include the following instance data:

- start date
- end date
- customer name
- project description
- address location of the project
- list of workers

The constructor for this class is setup to take in the project name, owner name, address of the project, and the start/end dates of the project. Since some projects might not have start and end dates assigned yet, there are two overloaded constructors for this class. Listing 18.1 shows the first part of the Project class, I have split it into pieces to better describe what is occurring in each section.

#### Listing 18.1 Project Class part 1

```

1 package lesson18_example;
2 import java.time.LocalDate;
3 import java.util.ArrayList;
4
5 public class Project implements Comparable { // #A
6 public ArrayList<Worker> workers = new ArrayList<>();
7 private String projectName, customer;
8 private Address projectAddress;
9 private double overheadPercent = .10, overheadAmount;
10 private LocalDate startDate, endDate; // #B
11
12 public Project(String projectName, String customer, Address projectAddress,
13 LocalDate startDate, LocalDate endDate){
14 this.projectAddress = projectAddress;
15 this.projectName = projectName;
16 this.startDate = startDate;
17 this.endDate = endDate;
18 this.customer = customer;
19 }
20
21 public Project(String projectName, String owner, Address projectAddress){ #C
22 this.projectAddress = projectAddress;
23 this.projectName = projectName;
24 customer = owner;
25 }

```

#A The project class implements comparable to compare the start dates of any two projects

#B Use the LocalDate class to store the start/end date of the project

#C This is considered an overloaded constructor since it does not have the same number of parameter values

**NOTE:** When I first started writing the main method for this application, I started using the Date class, but quickly realized that parts of the Date class are deprecated (which means they are no longer supported). Instead, I switched to using the java.time package and the LocalDate class to store the dates used in this application. I recommend checking out the JavaDoc from Oracle on the java.time package which is part of the Java API located here: <https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html>



Next, I have included a method to add the workers to the project using an ArrayList so I can easily add/remove/update the workers assigned to this project. There is also a method to calculate the cost based on the workers payroll plus any overhead amount for the general contractor. This general contractor does not earn an hourly wage, instead she assigns an overhead percentage which is added to the total cost for the project. This covers her time and expenses. The overhead amount is normally 10%, but the application has the ability to change that value depending on the degree of difficulty of the project.

### Listing 18.1 Project Class part 2

```

26 public String getName(){return projectName;} //A
27 public double getOverhead(){return overheadAmount;} //A
28 public double getTotalCost(){return calculateProjectCost();} //B
29 public void setOverhead(double overhead){ //C
30 this.overheadPercent = overhead;
31 }
32 public void addWorkers(ArrayList<Worker> workers){ //D
33 this.workers = workers;
34 }
35 public double calculateProjectCost(){
36 double totalCost = 0;
37 for(Worker w:workers) { //E
38 totalCost += w.calculatePay()
39 }
40 overheadAmount = overheadPercent * totalCost; //F
41 totalCost += overheadAmount;
42 return totalCost; //G
43 }

```

#A These methods return instance data to the calling program

#B This method calls the calculateProjectCost method and returns the project total cost

#C This method allows the overhead percentage to be changed

#D Assign the ArrayList from the calling program to the ArrayList in the Project class

#E Use an enhanced for loop to traverse the workers ArrayList

#F Calculate the overhead percent

#G Return the total cost for all the workers, materials and overhead percentage

Notice in the method to calculate the Project cost, I used an enhanced for loop to obtain the information about all the workers in the ArrayList. This is helpful if you are not sure how many workers are assigned to each project, the code loops through the ArrayList and stops when it reaches the last worker.

### Listing 18.1 Project Class part 3

```

45 @Override
46 public int compareTo(Object o) { //A
47 if(o instanceof Project) { //B
48 if(((Project) o).startDate.isAfter(this.startDate)) //B
49 return -1; //B
50 } //B
51 return 1; //B
52 }
53 @Override

```

```

54 public String toString(){ //#C
55 String projectDetails;
56 projectDetails = "Project name: %s" + "\nStart Date: " +
57 startDate + "\nCustomer: " +
58 customer + "\nAddress: " +
59 projectAddress.toString() + "\n" +
60 "=====\n";
61 for(Worker w: workers) {
62 projectDetails += w.toString() + "\n\n";
63 }
64 projectDetails += "Project Total: $%.2f\n";
65 projectDetails += "Contractor Overhead: $%.2f\n";
66 return projectDetails;
67 }
68 }
69 }

```

**#A** This is the overridden `compareTo` method that compares the two start dates

**#B** The if statement can be written in a more condensed version:

```
return ((Project) o).startDate.compareTo(this.startDate);
```

**#C** This method overrides the `Object toString` method to return a `String` with all the project details

In the project class, notice that it contains two overridden methods. The first is the `compareTo` method, this is required when you implement the `Comparable` interface from the Java API. In this class, I have provided logic to compare the start date of any two projects. In the `compareTo` method, the project compares the start date of the project that is invoking the method and passing the other project as an argument to the `compareTo` method.

By implementing the `compareTo` method in the `Project` class, it allows me to easily compare two projects. For this example, I am comparing the start dates of each project, but this method can easily be updated to compare other components of the project. Just remember that the `compareTo` method returns an integer value with the following values:

- negative number if the first object comes before the second object
- zero if they are equal
- positive number if the first object comes after the second object

This gives the programmer control on what is compared and what determines the order. We could change our project to compare two projects and identify which client name comes first alphabetically for example.

The second overridden method is the `toString` method. Remember, every class automatically inherits from the `Object` class which has a `toString` method. If I did not provide an overridden method, any time I called the `toString` method it would simply return the reference address of the object. Instead, in the `Project` class, the overridden version of the `toString` method formats and returns all the information about a project for printing.

## 18.1.2 The Address Class

The `Address` class is used to create an address object for either the worker or the location of the project. It is designed to allow for two types of addresses, one with only one street line

address, and a second constructor for addresses with a second street line. The second street line is often used for identifying an apartment number or even a PO Box.

### Listing 18.2 Address Class

```

1 package lesson18_example;
2 public class Address {
3 private String street1, street2, city, state;
4 private String zip;
5
6 public Address(String street1, String city, String state,
7 String zip){
8 this.street1 = street1;
9 this.city = city;
10 this.state = state;
11 this.zip = zip;
12 }
13 public Address(String street1, String street2, String city, String state, // #A
14 String zip) {
15 this.street1 = street1;
16 this.street2 = street2;
17 this.city = city;
18 this.state = state;
19 this.zip = zip;
20 }
21 @Override
22 public String toString(){ // #B
23 if(street2 == null) return street1 + "\n" +
24 city + ", " + state + " "+zip;
25 else return street1 + "\n" + street2 + "\n" +
26 city + ", " + state + " "+zip;
27 }
28 }

```

#A The Address class has an overloaded constructor for any address objects with a second street line

#B The toString method is overridden to return the address in a formatted String

### 18.1.3 The Worker class

Now that we have the Project class created, the next class I will add is the Worker class and it contains all the general information about a worker. This class will act as a superclass to the electrician, plumber and carpenter classes. Both the doWork() and the toString methods will be overridden in each subclass, these are examples of polymorphism.

### Listing 18.3 Worker Class

```

1 package lesson18_example;
2 public class Worker {
3 public String fName, lName;
4 public Address address;
5 public int idNumber;
6 public double hoursWorked;
7 public double hourlyRate;
8 public String doWork(){ // #A
9 return "Worker";

```

```

10 }
11 public Worker(String fName, String lName, Address address,
12 int idNumber, double hours, double rate) {
13 this.fName = fName;
14 this.lName = lName;
15 this.address = address;
16 this.idNumber = idNumber;
17 this.hoursWorked = hours;
18 this.hourlyRate = rate;
19 }
20 public void setHoursWorked(double hours) {
21 hoursWorked = hours;
22 }
23 public void setHourlyRate(double rate) {
24 hourlyRate = rate;
25 }
26 public double calculatePay() {
27 return hoursWorked * hourlyRate;
28 }
29 @Override
30 public String toString() { // #B
31 return fName + " " + lName + " \nCompensation: $" + calculatePay();
32 }
33 }

```

#A The `doWork()` method will be overridden by each of the subclasses of `Worker`

#B The `toString` method is overridden to return the name and salary for each worker in a formatted `String`

### 18.1.4 The Carpenter Class

The `Carpenter` class extends the `Worker` class. One of the differences between the superclass `Worker` and the `Carpenter` class is that it allows for the additional cost for lumber materials. It also overrides the `doWork()` and `toString()` methods which is an example that enables polymorphism. This happens when a `Worker` object is assigned a reference type of a `Carpenter` object and then invokes any of the overridden methods. The JVM first attempts to invoke the method in the `Carpenter` class, if it is not found, then it looks for the method in the superclass.

#### Listing 18.4 Carpenter Class

```

1 package lesson18_example;
2 public class Carpenter extends Worker { // #A
3 private double lumberCosts;
4 public Carpenter(String fName, String lName, Address address,
5 int idNumber, double hours, double rate){
6 super(fName, lName, address, idNumber, hours, rate); // #B
7 }
8 public void setLumberCosts(double amount){
9 lumberCosts = amount;
10 }
11 public String doWork(){ // #C
12 return "Complete carpentry work";
13 }
14 @Override
15 public String toString() { // #D

```

```

16 return "Carpenter: " + super.toString() + "\n" + doWork();
17 }
18 public double calculatePay(){ // #E
19 return hoursWorked * hourlyRate + lumberCosts;
20 }
21 }

```

**#A** The Carpenter class extends the Worker class which enables it to use the public attributes of the Worker class

**#B** This statement uses the Worker constructor to initialize the name, address, id number, hours and rate

**#C** The doWork() method is overridden, this is allowing the Carpenter to do specific carpentry work

**#D** The toString method is overridden to return the Carpenter information in a formatted String

**#E** Since the Carpenter might have additional lumberCosts, the calculatePay method is also overridden

### 18.1.5 The Electrician Class

The Electrician class also extends the Worker class. For this activity, I've included a variable to allow for additional expenses such as associated wiring costs. Since the Electrician class extends the Worker class, the constructor takes in the values for name, address, id number, hours and rate and uses the keyword super to assign these values to the instance data. This can be helpful especially if we need data validation, for example, if we want to make sure the hours and rate are not negative. This logic can be included once in the Worker class and then each subclass benefits from this data validation.

#### Listing 18.5 Electrician Class

```

1 package lesson18_example;
2 public class Electrician extends Worker{ // #A
3 private double wiringCost = 0.0;
4 public Electrician(String fName, String lName, Address address,
5 int idNumber, double hours, double rate){
6 super(fName, lName, address, idNumber, hours, rate); // #B
7 }
8 public void setWiringCosts(double amount){
9 wiringCost = amount;
10 }
11 public String doWork(){ // #C
12 return "Install electrical components";
13 }
14 @Override
15 public String toString() { // #D
16 return "Electrician: " + super.toString() + "\n" + doWork();
17 }
18 public double calculatePay(){ // #E
19 return hoursWorked * hourlyRate + wiringCost;
20 }
21 }

```

**#A** The Electrician class extends the Worker class which enables it to use the public attributes of the Worker class

**#B** This statement uses the Worker constructor to initialize the name, address, id number, hours and rate

**#C** The doWork() method is overridden, this is allowing the Electrician to do specific electrical work

**#D** The toString method is overridden to return the Electrician information in a formatted String

**#E** Since the Electrician might have additional wiringCosts, the calculatePay method is also overridden

### 18.1.6 The Plumber Class

The Plumber class also extends the Worker class. There are definitely more differences between these types of workers, but to keep the code to a minimum I've tried to keep it simple. So, the Plumber class is very similar to the Carpenter and Electrician classes. But I've added a variable for any additional costs associated with plumbing materials.

#### Listing 18.6 Plumber Class

```

1 package lesson18_example;
2 public class Plumber extends Worker{ //#A
3 private double plumbingMaterials = 0;
4 public Plumber(String fName, String lName, Address address,
5 int idNumber, double hours, double rate){
6 super(fName, lName, address, idNumber, hours, rate); //#B
7 }
8 public void setPlumbingCost(double amount){
9 plumbingMaterials = amount;
10 }
11 public String doWork(){ //#C
12 return "Install plumbing";
13 }
14 @Override
15 public String toString() { //#D
16 return "Plumber: "+super.toString() + "\n" + doWork();
17 }
18 public double calculatePay(){ //#E
19 return hoursWorked * hourlyRate + plumbingMaterials;
20 }
21 }
```

**#A** The Plumber class extends the Worker class which enables it to use the public attributes of the Worker class

**#B** This statement uses the Worker constructor to initialize the name, address, id number, hours and rate

**#C** The doWork() method is overridden, this is allowing the Plumber to do specific electrical work

**#D** The toString method is overridden to return the Plumbing information in a formatted String

**#E** Since the Plumber might have additional plumbing costs, the calculatePay method is also overridden

A future version of this application might include more information for each type of worker. There are definitely more differences between each type of worker, but this gives you an idea of how the project can be started.

## 18.2 Main Application

The main application for this contractor application is used to do the following:

- Create Address objects for two customers
- Using the LocalDate class from the java.time package from the java API to create date objects for the start/end dates for each project
- Create two projects, one for a new house and one for a small project to add outdoor motion lighting
- Create Address objects for all the workers, the last worker uses an overloaded

constructor since this worker has a second address line for the apartment number

- Add three workers, an electrician, plumber, and carpenter that can be used on these projects
- Add these workers to an ArrayList so we can assign them to a project
- Add all three workers to the house project
- Set the lumber costs to \$2000
- Set the general contractor overhead to 18% for this large project
- Print out the project information
- Repeat the process for the small outdoor motion lighting project
- Compare the start dates to determine which project needs to start first and print the appropriate message

Now that all the classes are created, we turn our attention to the main method. The goal of the main method in this example includes logic to demonstrate all the topics covered in this unit. A few important concepts are shown in Listing 18.7 (lines 28-30) where the Worker class is used to create a reference variable that points to a subclass object. These lines are enabling polymorphism.

Since all the workers are created using the superclass Worker, line 37 shows how I had to cast the worker as the specific subclass that was used when it was created to allow access to methods defined in the subclass that were not overridden methods.

Before reviewing the code for the main method for this application, here is a printout of some sample output:

```

Project name: House
Start Date: 2019-11-03
Customer: Shira Gotshalk
Address: 123 Main Street
Anywhere, PA 19001
=====
Carpenter: Yusef Eberly
Compensation: $2522.0
Complete carpentry work

Electrician: Peg Fisher
Compensation: $300.0
Install electrical components

Plumber: Harley Davidson
Compensation: $500.0
Install plumbing

Project Total: $3919.96
Contractor Overhead: $597.96

Project name: Motion Lights
Start Date: 2019-06-26
Customer: Maggie Thygeson
Address: 44 South Main Street
Cleveland, OH 42111
=====
Electrician: Peg Fisher
Compensation: $700.0
Install electrical components

Project Total: $770.00
Contractor Overhead: $70.00

The outdoor light project is scheduled before the addition

```

Figure 18.2 Screenshot of sample output for Contractor application

Listing 18.7 part 1 shows the main method that includes the code to create all the Address, Project, LocalDate, and Worker objects.

#### Listing 18.7 The main method for this application (part 1)

```

1 package lesson18_example;
2 import java.time.LocalDate;
3 import java.util.ArrayList;
4 import java.util.Date;

```



```

5
6 public class Lesson18_Example {
7 public static void main(String[] args) {
8 Address client1 = new Address("123 Main Street", "Anywhere", "PA", //#A
9 "19001");
10 Address client2 = new Address("44 South Main Street", "Cleveland", "OH",
11 "42111");
12 LocalDate start1 = LocalDate.parse("2019-11-03"); //#B
13 LocalDate end1 = LocalDate.parse("2020-05-29");
14 Project p1 = new Project("House", "Shira Gotshalk", client1, //#C
15 start1, end1);
16 LocalDate start2 = LocalDate.parse("2019-06-26");
17 LocalDate end2 = LocalDate.parse("2019-07-27");
18 Project p2 = new Project("Motion Lights", "Maggie Thygeson",
19 client2, start2, end2);
20
21 Address eAddress = new Address("467 Seminole Avenue", "Jenkintown",
22 "PA", "19446");
23 Address cAddress = new Address("88 Stallion Circle", "Horsham",
24 "PA", "19022");
25 Address pAddress = new Address("9821", "Apt B", "Siglerville",
26 "PA", "19345");
27
28 Worker e = new Electrician("Peg", "Fisher", eAddress, 1234, 15, 20); #D
29 Worker c = new Carpenter("Yusef", "Eberly", cAddress, 2456, 17.40, 30);
30 Worker p = new Plumber("Harley", "Davidson", pAddress, 3214, 25, 20);

```

#A The project address is required when creating a new project, so it must be created first

#B Using the `LocalDate` class from the `java.time` package which was added in Java 8

#C Create a new project for building a house

#D Create three worker objects: an electrician, carpenter, and a plumber

Next is the second part of the main method. This is where the workers are added to the project. The first project uses all three types of workers and extra costs are assigned for each worker. The second project only requires an electrician, so the plumber and carpenter are removed from the list of workers and the wiring costs are updated for the electrician prior to creating the project. Finally, the projects are printed using the overridden method `toString`.

#### Listing 18.7 The main method for this application (part 2)

```

31 ArrayList<Worker> workers = new ArrayList<>(); //#E
32
33 workers.add(c); //#F
34 workers.add(e);
35 workers.add(p);
36 ((Carpenter)c).setLumberCosts(2000); //#G
37 ((Electrician)e).setWiringCosts(3200);
38 ((Plumber)p).setPlumbingCosts(2750);
39 p1.addWorkers(workers); //house requires all three workers //#H
40 p1.setOverhead(.18); //the overhead is higher for a house
41
42 System.out.printf(p1.toString(), p1.getName(),
43 p1.getTotalCost(), p1.getOverhead());
44 System.out.println("*****\n");
45 + "\n*****\n");

```

```

46
47 workers.remove(p); //project 2 does not need a plumber //#I
48 workers.remove(c); //project 2 does not need a carpenter
49 ((Electrician)e).setWiringCosts(300); //#J
50 workers.get(0).setHoursWorked(20); //set the hours worked to 20 //#K
51 p2.addWorkers(workers);
52 System.out.printf(p2.toString(), p2.getName(),
53 p2.getTotalCost(), p2.getOverhead());
54
55 if (p2.compareTo(p1) < 0) {
56 System.out.println("\nThe "+p2.getName()+" project is "
57 + "scheduled before "+p1.getName());
58 } else {
59 System.out.println("\nThe "+p1.getName()+" project is "
60 + "scheduled prior to "+p2.getName()); }
61 System.out.println("\n");
62 p2.printPayroll();
63 }
64 }

```

**#E** Create an ArrayList to hold the workers for a specific project

**#F** Add all three workers to the list

**#G** Set the lumber costs for the carpenter, notice that it the worker object must be cast as a Carpenter to access this method

**#H** Add the list of workers to the house project

**#I** For the second project, only the electrician is needed, so remove the plumber and carpenter from the list

**#J** Set the wiring costs for the electrician, notice that it the worker object must be cast as a Electrician to access this method

**#K** Update the hours worked for the electrician, which is the only worker left in the ArrayList, so it is at position zero

**#L** Compare the start dates of the two projects to print out which project will start first

## 18.3 Summary

In this lesson, you learned:

- How to write an application that uses overloaded methods specifically for constructor methods
- How to create overridden methods in a subclass
- How to create a list of superclass objects that are instantiated as specific subclasses
- Based on the type of worker, update specific values for each project such as lumber costs, wiring costs by casting the superclass as one of the subclasses
- How to use the Comparable method and overriding the compareTo method to compare the start date of any two projects

This lesson provided an activity that included all the topics from lessons 13 - 17. In the next unit, I will continue to work with classes and objects. The first lesson in the next unit is all about the difference between pass by value vs. pass by reference.

### **Try this:**

Use the Contractor application and add the required code to print out a payroll statement by project that includes:

- Worker name, id, address, and compensation
- Include the project name

Figure 18.3 shows a sample of the output from print a payroll report.

```

Payroll Report for Project: House
(Salary only)
Carpenter: Yusef Eberly
ID number: 2456
88 Stallion Circle
Horsham, PA 19022
Hourly Rate: 30.0
Hours worked: 17.4
Total Compensation: $522.0
=====

Electrician: Peg Fisher
ID number: 1234
467 Seminole Avenue
Jenkintown, PA 19446
Hourly Rate: 20.0
Hours worked: 15.0
Total Compensation: $300.0
=====

Plumbing costs: Harley Davidson
ID number: 3214
9821
Apt B
Siglerville, PA 19345
Hourly Rate: 20.0
Hours worked: 25.0
Total Compensation: $500.0
=====

Payroll Report for Project: Motion Lights
(Salary only)
Electrician: Peg Fisher
ID number: 1234
467 Seminole Avenue
Jenkintown, PA 19446
Hourly Rate: 20.0
Hours worked: 20.0
Total Compensation: $400.0
=====

```

Figure 18.3 screenshot of payroll report

**Solution to the Try This activity:****18.8 Print the payroll by project - Project Class**

```

1 package lesson18_example;
2 import java.time.LocalDate;
3 import java.util.ArrayList;
4
5 public class Project implements Comparable {
6 //Please see Listing 18.1 for the omitted code //#A
7 public void printPayroll(){
8 System.out.println("Payroll Report for Project: " + getName());
9 System.out.println("Salary only");
10 for(Worker w: workers)
11 {
12 if(w instanceof Plumber) //#B
13 System.out.print("Plumbing costs: ");
14 else if (w instanceof Electrician)
15 System.out.print("Electrician: ");
16 else if(w instanceof Carpenter)
17 System.out.print("Carpenter: ");
18 System.out.println(w.fName + " " + w.lName + "\n"+
19 "ID number: "+w.idNumber + "\n"+
20 w.address.toString()+
21 "\nHourly Rate: " + w.hourlyRate +
22 "\nHours worked: " + w.hoursWorked +
23 "\nTotal Compensation: $" + w.hourlyRate * w.hoursWorked +
24 "\n=====+"\n");
25 }
26 }
27 }

```

#A To reduce the amount of space required, please refer to listing 18.1 for the omitted code

#B The method loops through all the worker objects, it then tests for a specific subclass to print the appropriate worker type

**18.9 Print the payroll by project – Main Class**

```

1 package lesson18_example;
2 import java.time.LocalDate;
3 import java.util.ArrayList;
4 import java.util.Date;
5
6 public class Lesson18_Example {
7 public static void main(String[] args) {
8 //Please see Listing 18.7 for the omitted code //#A
9 System.out.println("\n\n");
10 house.printPayroll(); //#B
11 // System.out.printf(house.toString(), house.getName(),
12 // house.getTotalCost(), house.getOverhead());
13 // System.out.println("*****")
14 // + "\n*****\n");
15
16 workers.remove(p); //project 2 does not need a plumber
17 workers.remove(c); //project 2 does not need a carpenter
18 ((Electrician)e).setWiringCosts(300);
19 workers.get(0).hoursWorked = 20; //set the hours worked to 20

```

```
20 project2.addWorkers(workers);
21 // System.out.printf(project2.toString(), project2.getName(),
22 // project2.getTotalCost(), project2.getOverhead());
23 //
24 // if (project2.compareTo(house) < 0) {
25 // System.out.println("\nThe outdoor light project is scheduled "
26 // + "before the addition");
27 // } else {
28 // System.out.println("\nThe addition is scheduled prior to the "
29 // + "outdoor lighting project");
30 // }
31 System.out.println("\n");
32 project2.printPayroll(); //#B
33 }
34 }
```

**#A** To reduce the amount of space required, please refer to listing 18.7 for the omitted code

**#B** Call the printPayroll method in the Project class

# ***Unit 4***

## ***More Programming with Objects***

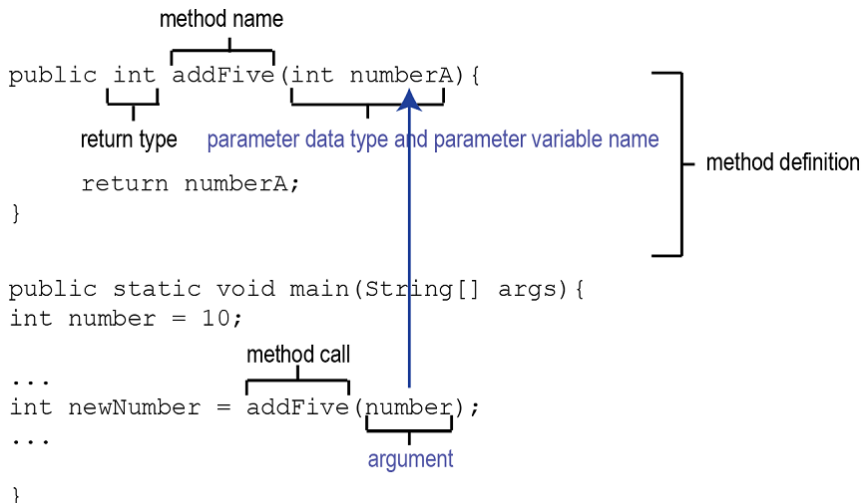
# 19

## Pass by Value vs. Pass by Reference

**After reading lesson 19, you will be able to:**

- Understand how Java passes arguments to parameters in a method
- Understand the impact of pass by value vs. pass by reference

In this lesson, you will learn that all arguments are copied into the parameter variables when a method is called. For this lesson, I want to have a consistent naming for the values described, so here is a diagram of the terms in this lesson:



**Figure 19.1** Diagram identifying the key terms for a method call

In figure 19.1, I have identified the key terms that are used when calling a method. Methods are designed to allow the calling program to pass in values as arguments. In this lesson, I will review scenarios that pass primitive data types, Strings and other objects. For each scenario, I will identify if it is a pass by value or pass by reference.

### **Consider This**

Most of the programs that are written use methods to help allow for code reuse and to make the program easier to read. When calling a method, it is important to understand the purpose of the method, what information does the method need and what information is returned.

We can pass information to a method; the original value is always copied to the parameter variables in the method header. In Java, the variable names can be identical between the calling program and the method parameter list. This does not cause a syntax error since Java allocates memory for each new variable in the method parameter list. These variables start with the same value as the calling program, but they can be changed by the method. When the method returns control to the calling program, the memory is released, and the variables are no longer accessible. In this case, the original values are unchanged.

It sounds like all variables are created equal, but it is important to remember that primitive data types have values stored with the variable name in memory, but object variables only store the memory address of the object.

So, when an object variable is copied to a parameter, the address is copied. Think of this as a house address. If we provide the address, it is possible to make changes to the house without updating the address. Then, when control is returned to the calling program, the house still has the changes.

But if we think of an example of a primitive data type, let's say quantity, and we start with a quantity of 5 oranges. That value is copied to our method where it might change the quantity to 6, but when the method returns control to the calling program, we still only have 5 oranges. That value is not changed.

## **19.1 Passing Primitive Types**

As I stated in the introduction, method arguments are either considered pass by value or pass by reference. If the data type of an argument is a primitive data type, then it is considered a pass by value. For primitive data types, the original argument value is never changed by the method. This lesson explains what happens to the argument values and parameter values before, during and after the method executes.

Let's look at an example. The output in figure 19.2 is from the code snippet in Listing 19.1.



```

Before calling the method
Value: 10.0
doubledValue: 0.0

After calling the method
Value: 10.0
doubledValue: 20.0

```

Figure 19.2 This figure is a screenshot of the output from executing the code in Listing 19.1

### Listing 19.1 Sample code that takes a number and returns the value of the number doubled

```

1 package lesson19_example;
2 public class Lesson19_sampleMethod {
3
4 public static double doubleMethod(double value) { // #B
5 value = value * 2; // #C
6 return value; }
7
8 public static void main(String[] args) {
9 double value = 10.0, doubledValue = 0.0; // #A
10 System.out.println("Before calling the method" +
11 "\nValue: " + value + "\ndoubledValue: " +
12 doubledValue + "\n");
13
14 doubledValue = doubleMethod(value); // #D
15
16 System.out.println("\nAfter calling the method" +
17 "\nValue: " + value + "\ndoubledValue: " +
18 doubledValue + "\n");
19 }

```

This method takes in a numeric value, doubles it and returns the new value

This statement prints the contents of value **before** it is used as an argument in the

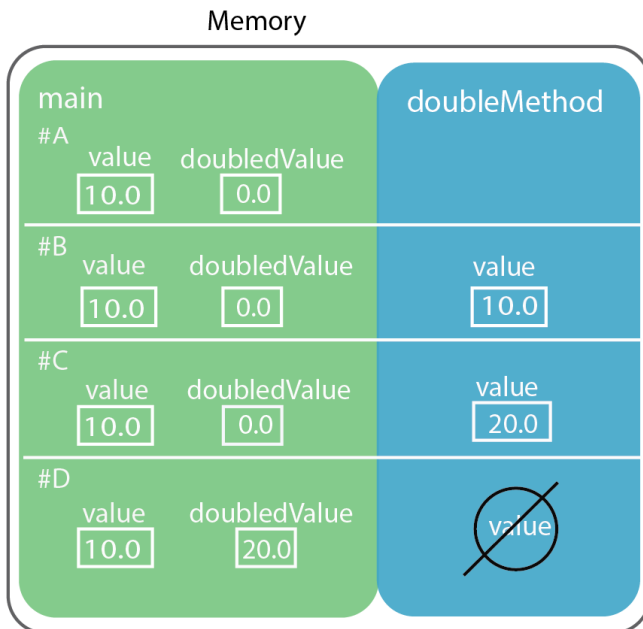
This statement prints the contents of value **after** it is used as an argument in the

Listing 19.1 shows the code snippet used to create the output in figure 19.2. The variable `value` from the main method starts as the value 10.0. When it is passed to the `doubleMethod`, the value of the argument variable on line 14 is copied into another double variable also called `value` on line 4. The method then uses the parameter variable and multiplies it by two, then returns the new value to the main method which is then stored in the variable `doubledValue`. In this example, I added print statements to the main method before and after the call to the `doubleMethod`. Once the method returns control to the main method, the parameter variable `value` is no longer accessible.

**NOTE:** In the example, the variable names of the argument and the parameter are both the same. When the program executes, the JVM creates **two** of the same variable names in memory with their respective values, it does NOT consider

these the same variable. The variable inside the `doubleMethod` is only available inside the method. It is considered a local variable that is only accessible to the code inside the method. After the return statement is completed, the variable is released from memory as shown in figure 19.3.

Figure 19.3 shows how the memory is updated for each of the labelled statements, #A, #B, #C and #D. Starting with line 9, two variables are created in memory and initialized with numeric data: `value = 10.0` and `doubledValue = 0.0`. Next, line 14 passes the value of 10.0 to the new parameter variable defined for the method on line 4, this appears in the `doubleMethod` section of the diagram. This value is multiplied by 2 and returned to the calling program on line 5. At that point, the value 20.0 is then placed in memory for the variable `doubledValue`. Once the return statement is executed, the method variable called `value` is no longer accessible and this is indicated in the diagram with a circle and strike through line.

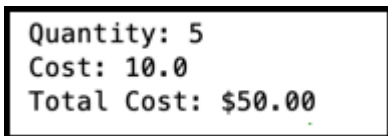


**Figure 19.3** This figure shows how the variables are created in memory and updated throughout the code snippet

Every argument value is copied into the corresponding parameter, but if it is a primitive data type the original argument value is never changed by the method. The method creates a local copy of the value that can be manipulated inside the method, when control is passed back to the calling method, this value is no longer available.

This process works the same for all primitive data types. Remember, a method can contain parameters of multiple primitive data types or even multiple parameters of the same data type. It can also contain a mix of primitive data types and object references. This section describes what happens to the variables defined as primitive data types. At the end of the lesson, I will review an example that has both primitive and object reference data types. Each time the method is called, the argument values are copied to the corresponding parameter variables. Once the method returns control to the calling method, the parameter variables are no longer accessible.

Before moving on to object reference types, let's look at one more example that has multiple parameter values. Figure 19.4 shows the sample output from executing the code snippet in Listing 19.2



```
Quantity: 5
Cost: 10.0
Total Cost: $50.00
```

Figure 19.4 is a screenshot showing the values of quantity, cost, and totalCost after executing the calculateCost method

#### Listing 19.2 Examples of pass by value using multiple parameters

```

1. package lesson19_example;
2. public class Lesson19_multipleParameters {
3. public static double calculateCost(int quantity, double cost) { //C
4. double total = quantity * cost; //D
5. quantity = 0; //E
6. cost = 0; //E
7. return total; //F
8. }
9.
10. public static void main(String[] args) {
11. double cost = 10.0, totalCost = 0.0; //A
12. int quantity = 5; //A
13. totalCost = calculateCost(quantity, cost); //B
14. System.out.println("Quantity: "+quantity);
15. System.out.println("Cost: "+cost);
16. System.out.printf("Total Cost: $%.2f\n",totalCost);
17. }
18. }
```

#A In the main method, declare and initialize three variables

#B Pass the argument values for quantity and cost to the calculateCost method

#C The method header receives two argument values and then copies them to the parameter values in the method

#D Calculate the total cost by multiplying quantity times cost

#E Set the quantity and cost to zero inside the method, this does not update the variables in the main method

#F Return the total cost to the calling program

In this example, the method name is `calculateCost` and it has two parameter variables for quantity and cost. The quantity variable is of type `int` and the cost variable is a `double`. Both of these variables are considered pass by value. The method multiplies the two values and returns the total cost to the calling program. The method also sets the values to zero after it calculates the total cost. Since all the values are pass by value, this does not change the values in the main method.

In the next section, I will explain what happens when the value is an object reference. In this case, the method copies the reference address and creates an alias to the location of the information for that object in memory. So, when the information is updated, and the method returns control to the calling program, the object values are still updated.

### Quick Check 19-1

#### Listing 19.3 Examples of pass by value

```

1. public class Lesson19_QuickCheck {
2. public static int increase(int a, int b) {
3. a = a + b;
4. return a; }
5. public static double decrease(int b) {
6. b = b - 1;
7. return b; }
8. public static double addTax(double total) {
9. total = total + total * .06;
10. return total; }
11. public static void main(String[] args) {
12. int counter = 0, start = 15;
13. double total = 10.0;
14. counter = increase(counter, start);
15. decrease(start);
16. total = addTax(total);
17. }
18. }
```

Using Listing 19.1 answer the following questions:

- 1) What is the value of counter after executing line 14?
  - a) 0
  - b) 15
  - c) 20
  - d) Nothing, there is a compile time error
- 2) What is the value of a on line 4?
  - a) 0
  - b) 15
  - c) 20
  - d) Nothing, there is a compile time error

- 3) What is the value of the variable `start` after executing line 15?
- a) 15
  - b) 14
  - c) 16
  - d) 0
- 4) What is the value of the variable `total` after executing line 16?
- a) 10.0
  - b) 10.60
  - c) 0.60
  - d) 0

## 19.2 Passing Reference Types

Next, let's review the second type of argument passing, pass by reference. Here is an example of a code snippet for pass by reference. Figure 19.5 is a screenshot of the output from the program in listing 19.4.

```
Employee before: Joey McQuiston Hourly Rate: 7.5
Employee after: Joey McQuiston Hourly Rate: 15.5
```

Figure 19.5 This figure shows the employee information before and after calling the `setHourlyRate` method.

### Listing 19.4 Create an Employee object from the Employee class and update the hourly rate

```
1. package lesson19_example;
2. class Employee {
3. public String fName, lName;
4. private double hourlyRate;
5. public Employee(String fName, String lName, double hourlyRate) {
6. this.fName = fName;
7. this.lName = lName;
8. this.hourlyRate = hourlyRate;
9. }
10. public String toString(){
11. return fName + " " + lName + " Hourly Rate: "+hourlyRate;
12. }
13. public void setHourlyRate(double hr) {
14. hourlyRate = hr; }
15. }
16. public class Examples {
17. public static void updateRate(Employee emp, double newRate) { //C
18. emp.setHourlyRate(newRate); } //D
19. public static void main(String[] args) {
20. Employee e = new Employee("Joey", "McQuiston", 7.50); //A
21. System.out.println("Employee before: " + e.toString());
```

```

22. updateRate(e, 15.50); //#B
23. System.out.println("Employee after: " + e.toString());
24. }
25. }
26.

```

**#A Create a new employee object**

**#B Pass the employee object and new rate to the updateRate method**

**#C UpdateRate method uses the employee object and new rate to change the employee hourly rate**

**#D Call the setHourlyRate method in the Employee class**

Pass by reference occurs any time the argument variable contains an object. The term pass by reference refers to the fact that the reference address of the object is copied from the argument value into the corresponding parameter value.

In this example, there is an Employee class that contains the first name, last name, and the hourly rate for each employee. In the main method, an Employee object is created and the reference address is stored in the variable `e` on line 21. The reference address to the Employee object is then passed to the `updateRate` method along with a new value for the hourly rate on line 23.

The `updateRate` method has two parameter variables, one for the reference value to the employee object called `emp` and a second parameter variable for the new hourly rate called `newRate`. Now that the `updateRate` method has a copy of the address for the employee object, it uses that address to find the memory location of the employee object and then it updates the hourly rate. When the method returns control to the calling program, the reference address to the employee object is not changed, but the hourly rate value stored at that reference is now updated to the new rate.

I also added print statements to show the value of the employee object before and after calling the method to update the hourly rate. Figure 19.5 shows the output from executing the code in Listing 19.4 and is reprinted here for easy reference.

```

Employee before: Joey McQuiston Hourly Rate: 7.5
Employee after: Joey McQuiston Hourly Rate: 15.5

```

**Figure 19.6** This figure shows the employee information before and after calling the `setHourlyRate` method.

This printout shows the values of the employee before and after calling the `update hourlyRate` method. Notice that the hourly rate is updated in the second print statement which occurs after the call to the `updateRate` method.

To help understand how pass by reference works, figure 19.6 shows a diagram of how the values are stored in memory. In this figure, an Employee object is created with a String for the first name and last name and a primitive data type for the hourly rate. The reference

address of the Employee object, `e` is copied to the parameter variable called `emp`. At this point in the program execution, `e` and `emp` are pointing to the same object, that is, the same location in memory. Next, the `updateRate` method changes the `hourlyRate` for the `emp` object to 15.50. When control is returned to the `main` method, the employee object, `e`, also shows the new rate since it refers to the same object in memory as `emp`.

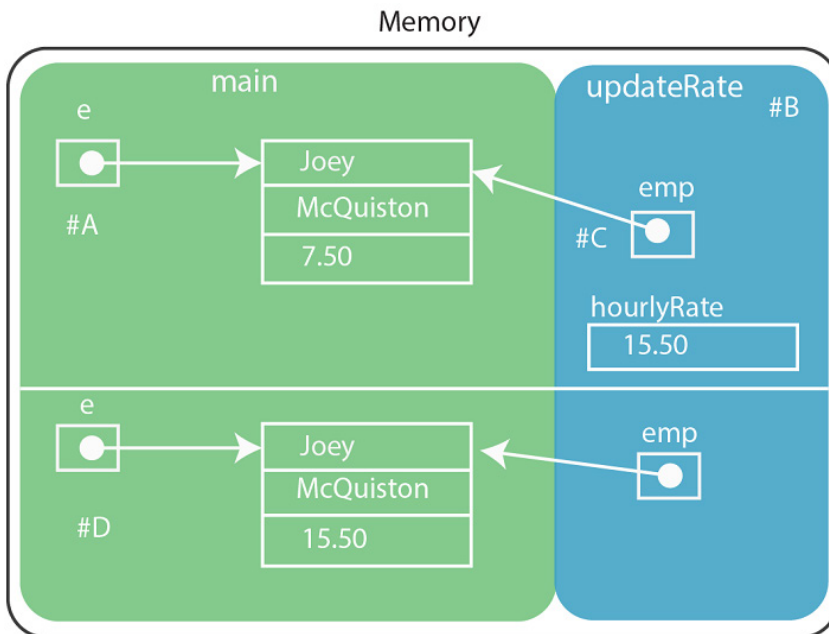


Figure 19.7 shows how the Employee object is updated by the call to the `setHourlyRate` method

In this example, an object was created and the reference address to the object was passed to a method. This is actually copying the address from the calling method argument to the parameter variable so they both contain the same value, the location of the object in memory. In our example, the location value is never changed, but if any of the values at that location are updated, the updates will persist when the method ends and returns control to the calling program. This can be confusing, so try to remember that arguments containing primitive data types cannot be altered by the method, but arguments that contain an object reference address can have the contents of the object updated the by the method.

### Quick Check 19-2:

#### Listing 19.5 Code for Quick Check 19-2

```
1. class Person{
```

```

2. String fName, lName;
3. public Person(String fName, String lName){
4. this.fName = fName;
5. this.lName = lName;
6. }
7. public void updateName(String s) {
8. this.fName = s;
9. }
10. public String toString(){
11. return fName + " " + lName;
12. }
13. }
14. public class Lesson19_QuickCheck2 {
15. public static void updateName(Person p, String newName) {
16. p.updateName(newName); }
17. public static void print(Person p) {
18. System.out.println(p.toString()); }
19. public static void main(String[] args) {
20. Person p = new Person("Mike", "Shepard");
21. print(p);
22. updateName(p, "Sue");
23. print(p);
24. }
25. }

```

Using Listing 19.5 answer the following questions:

1. **What is printed after executing line 21:**

- a. Mike Shepard
- b. Sue Shepard
- c. Null

2. **What is printed after executing line 23:**

- a. Mike Shepard
- b. Sue Shepard
- c. Null

## 19.3 Summary

In this lesson, you learned:

- What happens when arguments are passed to method parameters
- The impact and scope of pass by value vs. pass by reference

This lesson explained the differences between pass by value and pass by reference. When calling a method, the program might need to pass a value to the method. If the value is a variable that contains a primitive data type, then the value is copied to a new variable defined in the method parameter list. If the value is a reference to an object, then the reference address of the object is copied to a new variable in the parameter list. The difference is that changes to the object persist after the method returns control to the calling program.



To update a variable defined with a primitive data type, the method must return the updated value and the calling program must then assign the value back to the original variable. In the next lesson, I will introduce the topic of garbage collection in Java. This topic also relates to the persistence of data during program execution and after program execution.

### Try this:

The code snippet in listing 19.6 is supposed to create a student record, then calculate the average GPA for a year and then print out the information. But every time the program runs, the GPA prints zero. For this exercise, identify where the problem is located and make the appropriate corrections.

#### Listing 19.6

```

1. public class ErrorExample {
2. public static void main(String[] args) {
3. Student s1 = new Student("Peg", "Fisher");
4. double gpa = 0;
5. findGPA(gpa, 3.2, 4.0, 3.4, 3.8);
6. s1.setGPA(gpa);
7. System.out.println(s1.firstName + " " + s1.lastName + " GPA: "+
8. s1.gpa);
9. }
10. public static void findGPA(double averageGPA, double semester1,
11. double semester2, double semester3, double semester4){
12. averageGPA = semester1 + semester2 + semester3 + semester4/4.0;
13. }
14. }
15. class Student {
16. String firstName;
17. String lastName;
18. double gpa = 0.0;
19. public Student (String fName, String lName) {
20. firstName = fName;
21. lastName = lName;
22. }
23. public void setGPA(double gpa) {
24. this.gpa = gpa;
25. }
26. }
```

### Quick Check 19.1 Solution

#### Listing 19.1 Examples of pass by value

```

1. public class Lesson19_QuickCheck {
2. public static int increase(int a, int b) {
3. a = a + b; }
4. public static double decrease(int b) {
5. b = b - 1;
6. return b; }
7. public static double addTax(double total) {
8. total = total + total * .06; }
9. public static void main(String[] args) {
```

```

10. int counter = 0, start = 15;
11. double total = 10.0;
12. counter = increase(counter, start);
13. decrease(start);
14. total = addTax(total);
15. }
16. }
```

Using Listing 19.1 answer the following questions:

- 1) What is the value of counter after executing line 14?
  - a) 0
  - b) 15**
  - c) 20
  - d) Nothing, there is a compile time error
- 2) What is the value of a after executing line 3?
  - a) 0
  - b) 15**
  - c) 20
  - d) Nothing, there is a compile time error
- 3) What is the value of the variable start after executing line 15?
  - a) 15**
  - b) 14
  - c) 16
  - d) 0
- 4) What is the value of the variable total after executing line 16?
  - a) 10.0
  - b) 10.60**
  - c) 0.60
  - d) 0

### Quick Check 19-2 Solution:

#### Listing 19.3 Code for Quick Check 19-2

```

1. class Person{
2. String fName, lName;
3. public Person(String fName, String lName){
4. this.fName = fName;
5. this.lName = lName;
6. }
7. public void updateName(String s) {
8. this.fName = s;
9. }
10. public String toString(){
```

```

11. return fName + " " + lName;
12. }
13. }
14. public class Lesson19_QuickCheck2 {
15. public static void updateName(Person p, String newName) {
16. p.updateName(newName); }
17. public static void print(Person p) {
18. System.out.println(p.toString()); }
19. public static void main(String[] args) {
20. Person p = new Person("Mike", "Shepard");
21. print(p);
22. updateName(p, "Sue");
23. print(p);
24. }
25. }

```

Using Listing 19.3 answer the following questions:

1. What is printed after executing line 21:

- a. Mike Shepard
- b. Sue Shepard
- c. Null

2. What is printed after executing line 23:

- a. Mike Shepard
- b. Sue Shepard
- c. Null

**Solution to the Try This activity:**

#### Listing 19.7 Solution to Try This Activity

```

1. public class ErrorExample {
2. public static void main(String[] args) {
3. Student s1 = new Student("Peg", "Fisher");
4. double gpa = 0;
5. gpa = findGPA(3.2, 4.0, 3.4, 3.8); //#A
6. s1.setGPA(gpa);
7. System.out.println(s1.firstName + " " + s1.lastName + " GPA: "+
8. s1.gpa);
9. }
10. public static double findGPA(double semester1, //#B
11. double semester2, double semester3, double semester4){
12. double averageGPA = semester1 + semester2 + semester3 + semester4/4.0;
13. return averageGPA; //#C
14. }
15. }
16. class Student {
17. String firstName;
18. String lastName;
19. double gpa = 0.0;
20. public Student (String fName, String lName) {
21. firstName = fName;

```

```
22. lastName = lName;
23. }
24. public void setGPA(double gpa) {
25. this.gpa = gpa;
26. }
27.
28. }
```

**#A** Add an assignment to the value returned from the method with the `gpa`

**#B** Change the method return type to `double` and remove the `gpa` parameter value

**#C** Add a return statement to return the `gpa`

# 20

## Garbage Collection

**After reading lesson 20, you will be able to:**

- Understand how Java allocates memory for objects using the Heap
- Understand how Garbage Collection (GC) works in Java

Even though memory has become less expensive and more compact, there are still physical limits to how much memory an application is allocated during runtime. Early versions of Java required the programmer to manually search for unused objects and remove them to recapture this space. In this lesson, I will explain how this process is now automated and why it is important.

### Consider This

Have you ever been in the middle of downloading a movie to your phone just to find out that you didn't have enough space? Your phone has some automatic deletion, but there are times when you need to manually delete content. If every app that you use during one day on your phone remained running, it would not take long for the memory to get used up and the battery to die.

In Java, there is a process to remove unused data called garbage collection. This process frees up memory and allows new objects to be added to the heap.

### 20.1 Java Heap

When a program creates objects, each object has a memory address that points to the location in memory of the instance data. This address is used to retrieve the data, update the data and even delete the data. The Java heap refers to a section of storage that is allocated for storing objects during program execution. The heap is actually split up into smaller parts. They include: Young Generation, Old or Tenured Generation, and Permanent Generation. All

objects start out in the young generation, specifically in eden. Figure 20.1 shows a picture of a sample heap in Java.

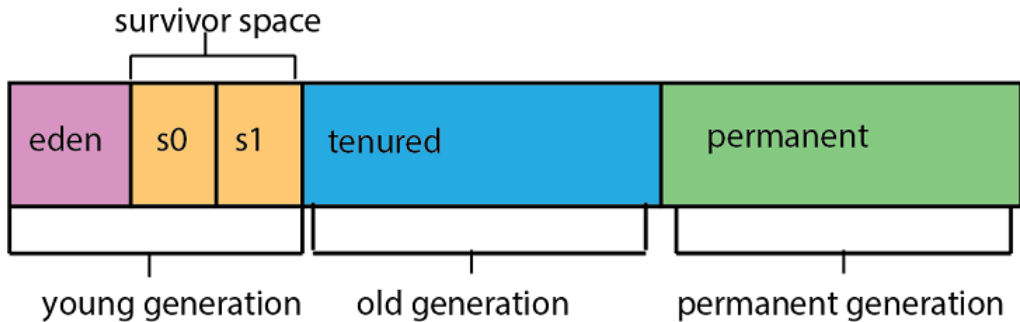


Figure 20.1 is a diagram of the Java heap memory allocation

So, the question is how does the JVM know when an object is eligible for deletion. In the next section, I will explain how the garbage collection process works.

### Quick Check 20-1:

1. What part of the Java heap is used when an object is initially created?
  - a. Eden
  - b. Survivor space
  - c. Tenured

## 20.2 Garbage Collection

The main objective of the Garbage Collector is to free heap memory by destroying unreachable objects. This process frees up memory space in the heap for new objects to be created. It is critical to manage memory efficiently and accurately to increase application performance and reliability. All objects start in the young generation space. When this area fills up, it triggers the JVM to perform a minor garbage collection. Garbage collection is the process of removing any references to objects that are no longer needed in your program. Since most Java objects have a short life span, this collection is very efficient at removing objects that are no longer needed. Objects that remain after the minor garbage collection are then aged and eventually move to the old generation.

The permanent generation storage area is used for metadata that is required by the JVM. It also contains the Java SE library classes and methods. When this area fills up it automatically triggers a major garbage collection. This process reviews all objects in the old generation and identifies any objects eligible for GC.

Although Java performs garbage collection automatically, it is increasingly important to identify objects that are no longer needed, and that they are eligible for garbage collection. An object is considered eligible for garbage collection when it is no longer reachable. These are some of the different ways to make an object eligible for garbage collection:

- Nullifying the reference variable
- Re-assigning the reference variable
- Creating an object inside a method
- Island of Isolation

### 20.2.1 Nullifying the reference variable

Consider an application that contains a list of students. Each student is created by providing the first name, last name, and age. If the purpose of creating each student object is only to print their information, once that process is done, the object can be made eligible for garbage collection. Let's look at a simple example:

#### Listing 20.1 Create a student object

```

1. package lesson20_listing1;
2. public class Lesson20_Listing1 {
3. public static void main(String[] args) {
4. Student s1 = new Student("Ana", "Gomez", 23); // #A
5. System.out.println(s1.getInfo());
6. s1 = null; // #B
7. }
8. }
9. class Student {
10. String firstName;
11. String lastName;
12. int age;
13. public Student(String fName, String lName, int age)
14. {
15. firstName = fName;
16. lastName = lName;
17. this.age = age;
18. }
19. public String getInfo() {
20. return firstName + " " + lastName + " age: "+age;
21. }
22. }
```

#A Create a student object

#B After the student information is printed, set the reference value to null

In this simple example, immediately after the object is created and then used in a print statement, the reference variable is set to null. This tells the JVM that the storage space on the heap that holds the values: [Ana, Gomez, 23] can be marked as available again. The values are not actually erased, but the memory is released for future use by other objects and the reference variable no longer has the memory address for this object. As a matter of fact, if

the program tries to print the information after setting the variable to null, it will generate an exception message: `java.lang.NullPointerException`. Figure 20.2 shows this process graphically.

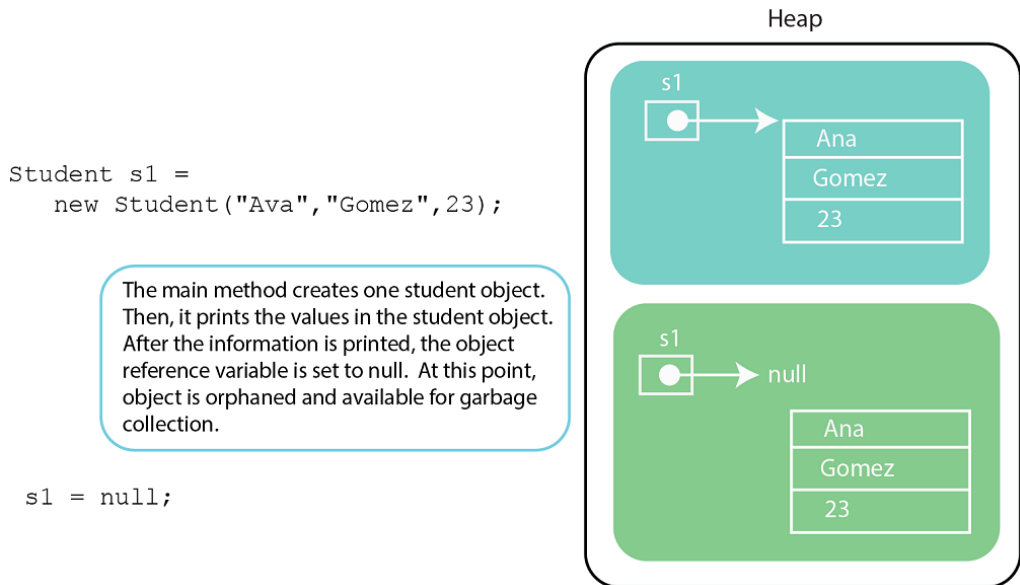


Figure 20.2 is a diagram of the heap in Java before and after setting an object to null

Notice in the diagram that the information for Ana Gomez is still in the heap after the variable `s1` has been set to null but there is no way to access the information. Once the variable reference is set to null, the object is available for garbage collection and the space can be reassigned.

## 20.2.2 Reassigning the reference variable

The next way that an object can be made eligible for garbage collection is when the reference variable is reassigned to another object. For example, if we create two reference variables to two separate objects, if the second reference variable is reassigned to the same value as the first reference variable, it now contains the reference address to the first object information. At this point, the information about the second object is no longer accessible and it is now eligible for garbage collection. Listing 20.2 shows the code for this example.

### Listing 20.2 Reassign a reference object

```
1. package listing2;
2. public class Lesson20_Listing2 {
3. public static void main(String[] args) {
```



```

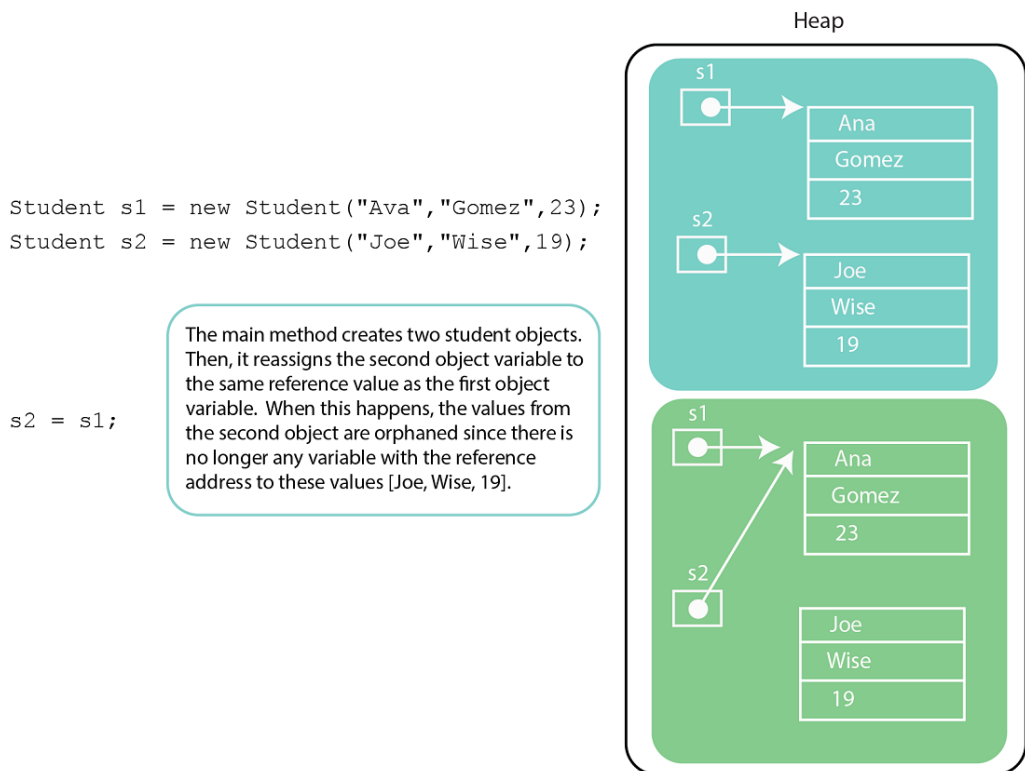
4. Student s1 = new Student("Ana", "Gomez", 23); //#A
5. Student s2 = new Student("Joe", "Wise", 19); //#A
6. s2 = s1; //#B
7. }
8. }
9. }

```

**#A Create two student objects**

**#B Assign the second student variable to the first object**

In this example, the first reference variable, `s1`, points to the student: [Ana, Gomez, 23] and the second variable, `s2`, points to the second student: [Joe, Wise, 19]. Next, the program reassigns the reference variable `s2` to the value of the reference variable `s1`. Now, both variables have the same memory address pointing to the values for the student: [Ana, Gomez, 23] and the information for Joe Wise is no longer accessible. Diagram 20.3 shows a visual representation of this example.



**Figure 20.3** is a diagram of the heap in Java before and after assigning an object reference variable to another object reference

In this diagram, the top portion shows how two reference variables are created and they each point to unique student objects with a different memory address for their respective student information. When the statement `s2 = s1` is executed, the second student reference variable is now pointing to the first student reference object. So, `s1` and `s2` both contain the same object reference address, therefore the second object is unreachable. Now the second object is eligible for garbage collection. In the bottom portion of the diagram, we can see that the second object no longer has any reference variables, so it is essentially removed.

### 20.2.3 Creating an object inside a method

Remember that all variables defined inside a method are only visible during the method execution. Once the method returns control to the calling program, any new object reference variables defined inside the method are no longer visible and become eligible for garbage collection (GC). If the method does not return a reference to the object and the object was not passed to the method as an argument, the object is eligible for GC. In addition, some of the objects might be eligible for GC if they no longer have any reference variables point to them.

In this example, the code creates two students in the main method. Then it calls the method `createNewStudent()`, to create a third student. The third student object is created inside this method, so when the method ends, the program does not have a reference to that object any more. So, the object and the reference variable are eligible for GC. In this example, lines 7 & 8 does not compile since the new student object (`student3`) is no longer visible. The error message says error: **cannot find symbol student3**.

#### Listing 20.3 Create an object inside a method

```

1. package listing3;
2. public class Lesson20_Listing3 {
3. public static void main(String[] args) {
4. Student s1 = new Student("Ana", "Gomez", 23); // #A
5. Student s2 = new Student("Joe", "Wise", 26); // #A
6. createNewStudent(); // #B
7. System.out.println(s1.getInfo() + "\n" + s2.getInfo() + // #C
8. student3.getInfo());
9. }
10. public static void createNewStudent(){
11. Student student3 = new Student("Isabella", "McKnight", 25);
12. }
13. }
14. class Student {
15. //the code for the student close is omitted to save space
27. }

```

#A Create two student objects

#B Call the method `createNew Student`

#C Print out the information for each student

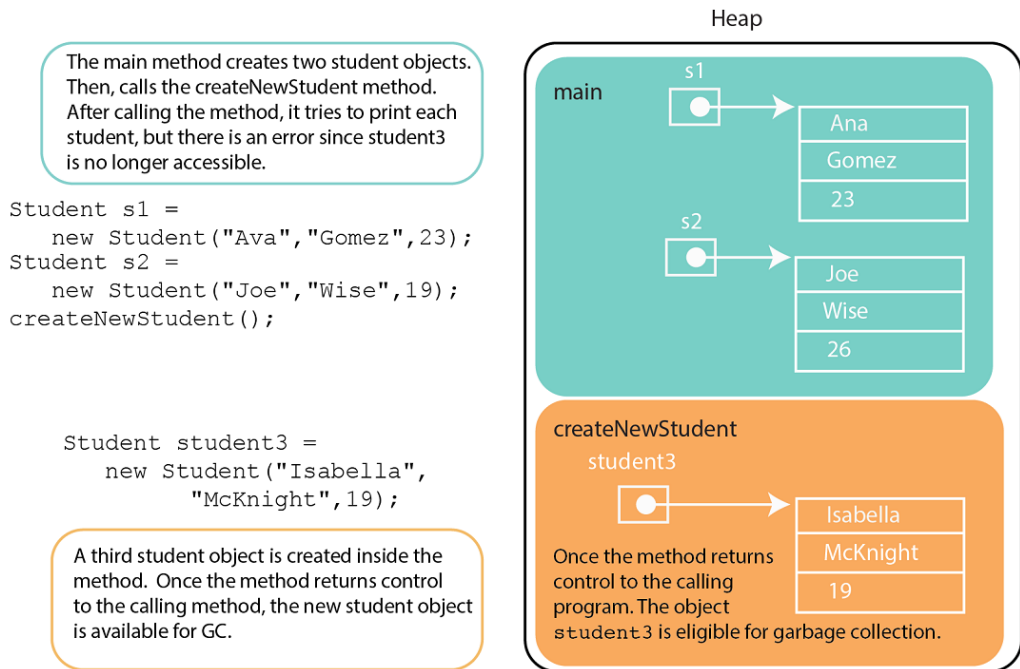


Figure 20.4 Shows how objects created in a method are eligible for GC after the method executes

In the figure 20.4, the bottom section is used to show the object that are created inside the createNewStudent method. The method creates a third object to hold information about a third student. Now, the variable student3 has the memory address for the student: [Isabella, McKnight, 19].

## 20.2.4 Island of Isolation

The last topic occurs less often, it is called the island of isolation. Put simply, this happens when two object references are assigned to each other, for example, Object 1 references Object 2 and Object 2 references Object 1. When this happens, neither Object 1 nor Object 2 is referenced by any other object and it causes an island of isolation. Listing 20.4 shows a simple example that causes this situation.

### Listing 20.4 Example of island of isolation

```
1. package listing4;
2. public class Address {
3. Address test; //#A
4. public static void main(String[] args) {
5. Address test1 = new Address(); //#B
6. Address test2 = new Address(); //#B
```

```
7. test1.test = test2; //#C
8. test2.test = test1; //#C
9. test1 = null; //#D
10. test2 = null; //#D
11. //Now the two test objects created on lines 7-8 are not accessible
12. }
13. }
```

**#A Create an Address object called test**

**#B Create two additional Address objects inside the main method**

**#C The test object for test1 is assigned to test2, and the test object for test2 assigned to test2**

**#D The two objects test1 and test2 are set to null**

In this code example, a new reference variable is created for an Address object called test. In the main method, two more Address reference variables are created (lines 5 and 6). When they are created, they contain an instance of the Address variable test since it is defined as part of the Address class. Next, the test variable for each object is set to the other variable reference value. Now, test1.test references to test2, and test2.test references test1. Finally, lines 9 and 10 are used to set test1 and test2 to null. So, at this point, the two test objects point to each other and there is no other reference variable to these objects. They are now considered an island of isolation and eligible for GC. Figure 20.5 shows a graphical representation of how an island of isolation can be created.

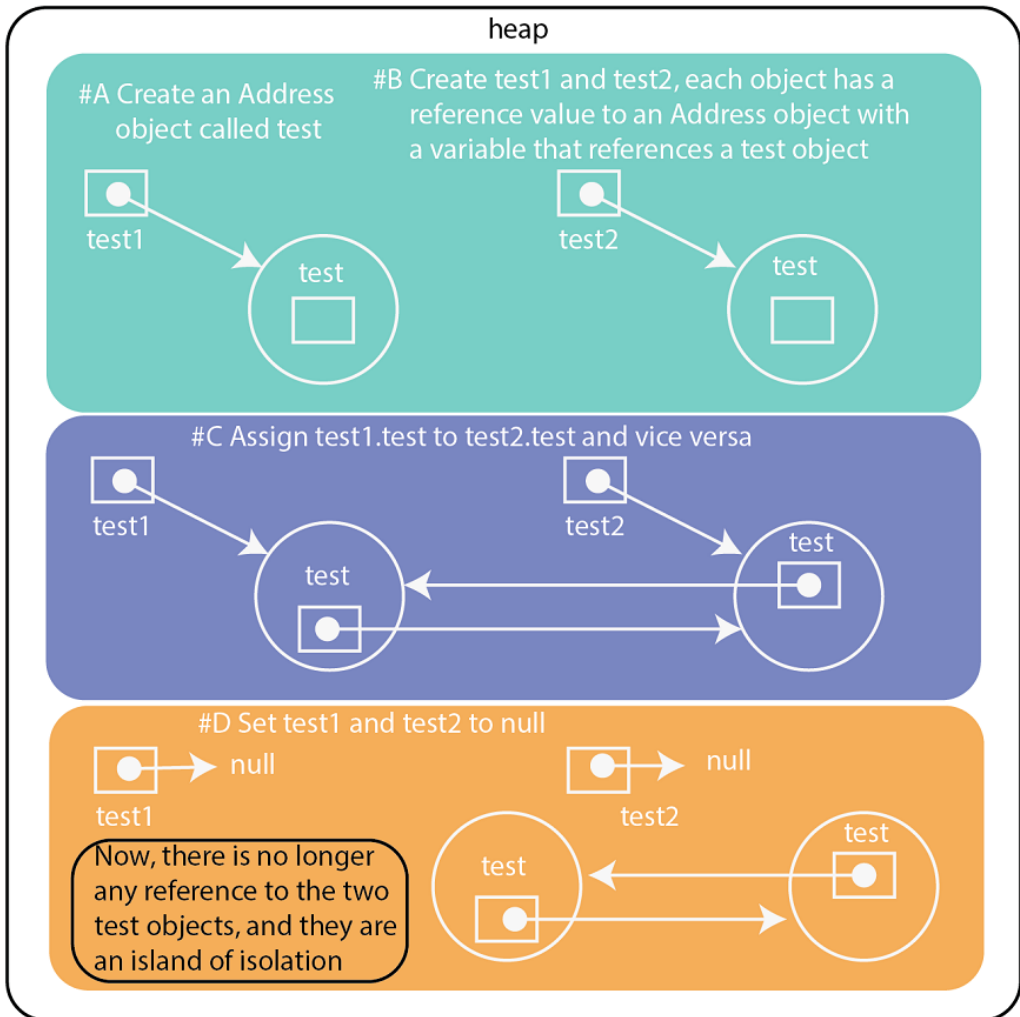


Figure 20.5 is a diagram of an example of how an island of isolation can occur

It is important to note that all application threads are stopped during the minor garbage collection process. This is often referred to as a Stop the World Event.

**Quick Check 20-2: Use this code listing to answer the questions below:**

**Listing 20.5 Sample code for quick check**

```
1. public class Lesson20_QuickCheck {
2. public static void main(String[] args) {
```

```

3. Student s = new Student("Evy", "Knapp", 18);
4. System.out.println(s.getInfo());
5. s = new Student("Lily", "Berkun", 20);
6. System.out.println(s.getInfo());
7. Student s1 = new Student("Aria", "Stark", 19);
8. System.out.println(s1.getInfo());
9. s1 = null;
10. //...
11. }
12. }
13. class Student {
14. //student class information goes here
15. }

```

1. **When is the student object that contains this information [Evy, Knapp, 18] eligible for GC?**
  - a. Line 3
  - b. Line 4
  - c. Line 5
  - d. After program execution is complete
2. **What line contains an example of reassigning an object reference?**
  - a. Line 3
  - b. Line 4
  - c. Line 5
  - d. After program execution is complete
3. **When is the object referenced by the variable s1 eligible for GC?**
  - a. Line 5
  - b. Line 7
  - c. Line 9
  - d. After program execution is complete

## 20.3 Summary

In this lesson, you learned:

- How Java allocates memory for objects using the Heap
- How Garbage Collection (GC) works in Java including the four ways to make objects eligible for GC

This lesson reviewed how the JVM manages allocating space on heap when new objects are created and then aged through the heap. All new objects start in the space labelled the young generation and then move to the old generation if they are still active.

This lesson also identified the processes that are used to make the objects eligible for GC. These include: nullifying an object, reassigning a reference variable, creating objects inside a method, and creating an island of isolation where the objects are no longer reachable.

**Quick Check 20-1 Solution:**

1. What part of the Java heap is used when an object is initially created?
  - a. Eden
  - b. Survivor space
  - c. Tenured

**Quick Check 20-2 Solution: Use this code listing to answer the questions below:****Listing 20.5 Sample code for quick check**

```

1. public class Lesson20_QuickCheck {
2. public static void main(String[] args) {
3. Student s = new Student("Evy", "Knapp", 18);
4. System.out.println(s.getInfo());
5. s = new Student("Lily", "Berkun", 20);
6. System.out.println(s.getInfo());
7. Student s1 = new Student("Aria", "Stark", 19);
8. System.out.println(s1.getInfo());
9. s1 = null;
10. //...
11. }
12. }
13. class Student {
14. //student class information goes here
15. }

```

1. When is the student object that contains this information [Evy, Knapp, 18] eligible for GC?
  - a. Line 3
  - b. Line 4
  - c. **Line 5**
  - d. After program execution is complete
2. What line contains an example of reassigning an object reference?
  - a. Line 3
  - b. Line 4
  - c. **Line 5**
  - d. After program execution is complete
3. When is the object referenced by the variable s1 eligible for GC?
  - a. Line 5
  - b. Line 7
  - c. **Line 9**
  - d. After program execution is complete





# 21

## Java Collections: List

**After reading this lesson, you will be able to:**

- Identify the interfaces and classes included in the Collections Framework
- Understand when to use the List collection
- Create lists using the ArrayList, LinkedList, and Vector classes
- Apply methods available for the List collection in a Java program

This lesson introduces the topic of collections in Java, and the types of collections available. After the overview, this lesson concentrates on the List interface. It includes the benefits of using and the difference between the types of Lists. A collection is an object that represents another group of objects. Often you will hear the term, a collections framework, this is an architecture for representing and manipulating collections.

### Consider This

A convenience store has a rack full of candy items. The rack is close to the register where the customers check out, so the items are frequently removed from the rack and need to be restocked. To keep a list of the items on the rack, we must consider a few key questions, such as can there be duplicate items, how often are items added or removed? In this lesson, you will see that this type of list fits the criteria for a LinkedList.

### 21.1 Collections in Java

Collections provide support for performing complex operations on small and large amounts of data such as searching, sorting, inserting, manipulating, and deleting elements. An element is the term used to reference a single object in a collection. The Collection framework includes interfaces and classes. There are four main interface types of Collections in Java, they are:

- List – contains an ordered set of objects that can have duplicates (note: ordered does not necessarily mean sorted), for example: [Jeep, Chevy, Ford, Jeep]

- Set – a collection of objects that does **NOT** contain any duplicates, for example: [Jeep, Chevy, Ford]
- Queue – a collection that uses a specific order for adding and removing elements, such as first in, first out (FIFO), for example: [customer3, customer2, customer1]
- Map – this type uses a map to associate unique keys to a value in the collection, therefore it does **NOT** allow duplicate keys, for example: (zipCode, cityName), (17084, "Reedsville")

Each interface in the framework has one or more classes that implement the interface. Figure 21.1 shows a diagram of the Java Collection Framework where you can see the interfaces and the classes that implement the interfaces. In this diagram, I have only listed a subset of all the available classes and interfaces, but these are the most widely used. In the next few lessons, I will review the four major interfaces: List, Set, Queue, and Map.

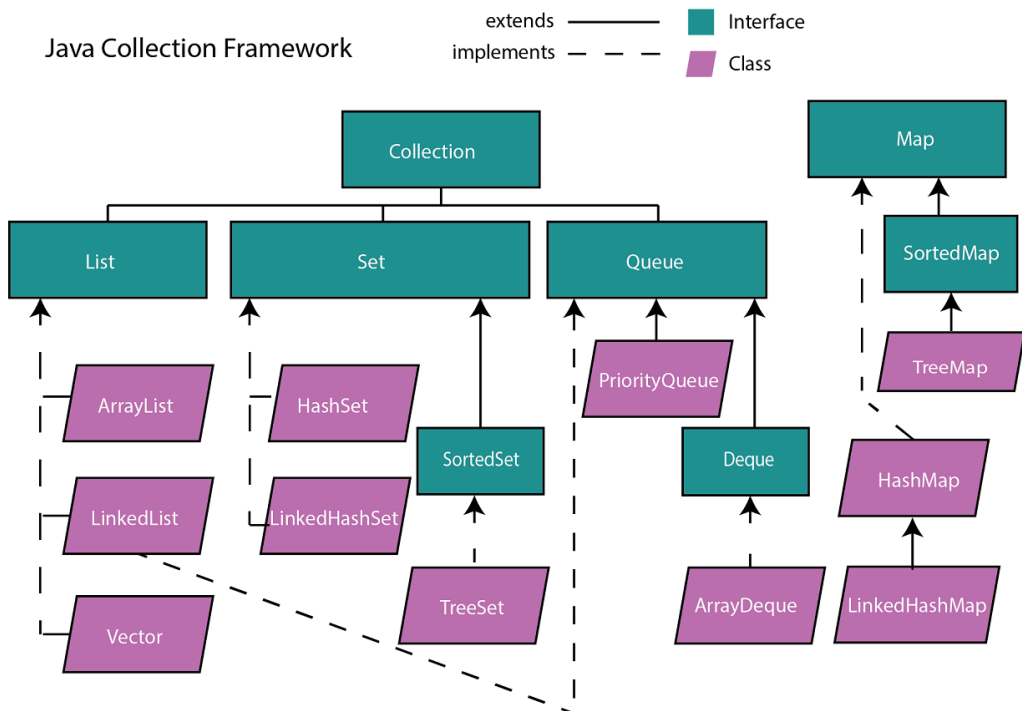


Figure 21.1 Diagram of the Java Collection Framework

In the diagram, the rectangles represent the interfaces that are part of the Collection framework including the Map interface. The Map interface does not directly extend the Collection framework, but it is still considered part of the overall Collections framework. List,

Set and Queue all extend the Collection interface, which means they inherit the methods defined in the Collection interface.

Remember, we can't instantiate an object as an interface, for example: `List l = new List<>()`, so each interface has classes that implement the respective interfaces. For example, ArrayList implements the List interface, so any object defined as an ArrayList automatically has access to the methods defined in the List interface: `List l = new ArrayList<>()`. Take a moment to familiarize yourself with the various interfaces and classes in the framework. Here are two useful links to the JavaDoc for the Collection Interface:

Java 8: <https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html>

Java

11:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Collection.html>

### Quick Check 21-1:

1. Identify the interface(s) that are implemented by the LinkedList Class:

- a. List
- b. Set
- c. Queue
- d. Map

2. Which class(es) can be used to complete this code snippet:

```
List list1 = new _____<String>();
```

- a. List
- b. Vector
- c. HashSet
- d. LinkedList

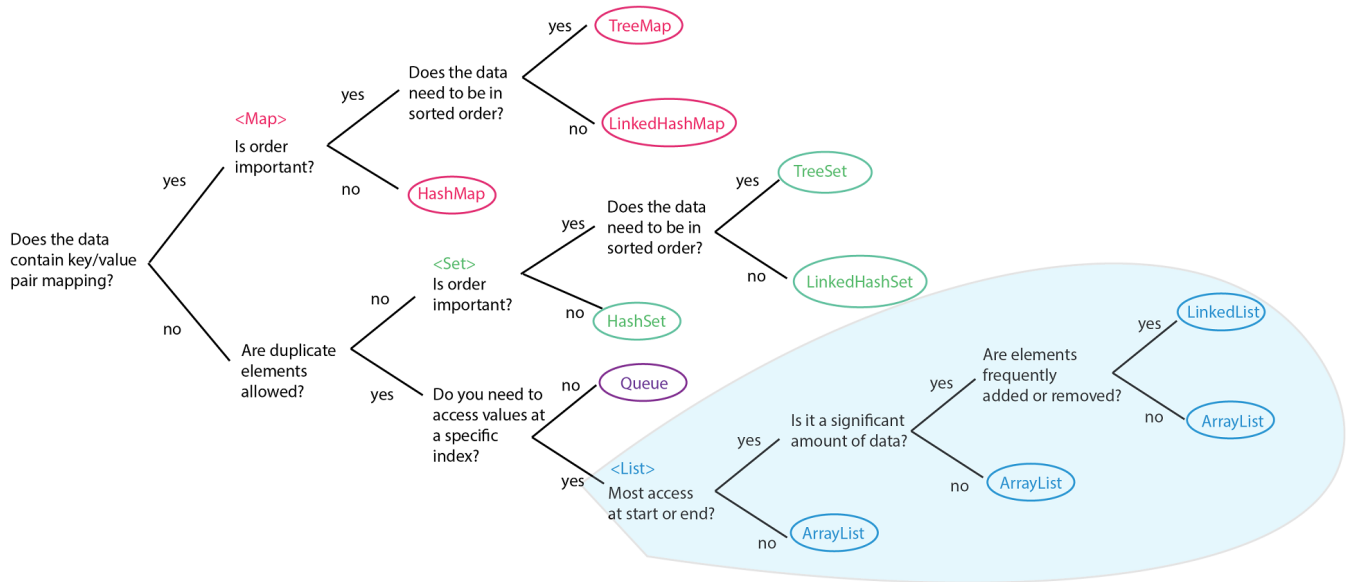
3. Which class(es) can be used to complete this code snippet:

```
Map<Integer, String[]> map1 = new _____<Integer, String[]>();
```

- a. Map
- b. SortedMap
- c. HashMap
- d. LinkedHashMap

## 21.2 Understand when to use the List collection

Choosing the right Collection can make a big difference in your application performance, readability, and reuse. To help decide which Collection to use, I've created a decision tree as seen in Figure 21.5. The section for the List interface is shaded.



**Figure 21.2** A decision tree for choosing the right Collection in Java

Follow each question by answering yes/no until you reach the appropriate collection type. For example, if we have a list of customers that have ordered items online, we can follow the decision tree to determine which type of collection to use to hold the list of customer orders.

This list does not contain a key/value pair, so next we check to see if duplicates are allowed. Since a customer may return and purchase more items, duplicates are allowed. Next, do we need to access elements at a specific index, for this example, we choose the yes path. We want to fill the orders on a first come, first serve basis, so we continue to the question about a significant amount of data. Again, we might have a high volume of sales and new customers are frequently added and removed once their order is filled, so we end up with a LinkedList. Here is a sample code snippet using a LinkedList:

```
List<String> customerList = new LinkedList<String>();
```

These are not strict rules that have to be followed, but this helps to choose the most efficient collection type for your application.

### Quick Check 21-2:

Use the decision tree to find the appropriate choice for these scenarios:

1. You have a collection of elements that need to be sorted in their natural order, and each element has a unique string associated with its value, which is best suited for your needs?

- a. `ArrayList`
- b. `HashMap`
- c. `HashSet`
- d. `TreeMap`

## 21.3 `ArrayList`, `LinkedList` and `Vector`

One important thing to remember when working with collections, is that you cannot instantiate an interface. For example, the `List` interface must create a new list as either an `ArrayList`, `LinkedList`, or `Vector`. When using a list, the user has control over where each element is added to the list, which element to delete, etc. The user can also access specific elements using their index value in the list. It even has a method to search for an element in the list.

A few key points about a `List` in Java:

- The `List` interface allows for duplicate elements
- Elements can contain null values
- The indexes start at position zero
- `List` supports Generics, which are discussed in Lesson 24

**Table 21.1** Shows each type of `List` and a brief description of their differences

| List Type               | Description                                                                                                                                                                                                                                                                                                                                               |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ArrayList</code>  | A list of elements that can grow and shrink as needed. These elements can be added, removed, or modified using an index value. When elements are removed, the list must adjust all remaining elements to fill the open spot. When the list is full, it automatically doubles the size and copies all current elements to the new <code>ArrayList</code> . |
| <code>LinkedList</code> | One difference between a <code>LinkedList</code> and an <code>ArrayList</code> is the way the elements are added to the list. A <code>LinkedList</code> maintains its insertion order. It is more efficient when adding/removing elements because it does not need to re-adjust the list.                                                                 |
| <code>Vector</code>     | A <code>Vector</code> is synchronized where the <code>ArrayList</code> and <code>LinkedList</code> are not. When the list is full, it automatically doubles the size and copies all current elements to the new <code>Vector</code> .                                                                                                                     |

We can create any of these list types with a variable declared as a `List` reference type since it is the super class for each of these subclasses. Here is a code snippet that uses an `ArrayList` to hold the color values of the rainbow:

### Code Listing 21.1 sample code to create a list of string objects for the colors of the rainbow

```
1. List<String> rainbow = new ArrayList<>();
2. rainbow.add("red");
3. rainbow.add("orange");
4. rainbow.add("yellow");
```

```

5. rainbow.add("green");
6. rainbow.add("blue");
7. rainbow.add("indigo");
8. rainbow.add("violet");

```

Lesson 6 has more detailed information on how to create and use ArrayLists since it is a commonly used data structure for storing lists of data. An ArrayList is considered a Collection since it implements the List interface.

**NOTE:** Unlike ArrayLists, arrays are not part of the Collections framework.

A LinkedList uses a doubly linked list to store the elements in the list. Let's start by looking at an example of a singly linked list that contains test scores for four students on a recent test:

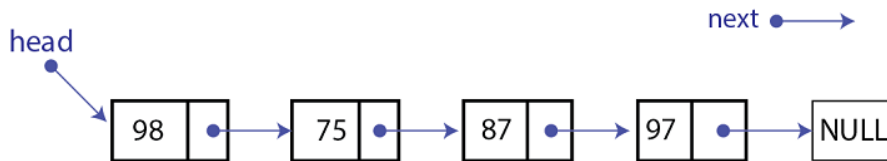


Figure 21.3 This is an example of a single linked list

In this example, each node in the list contains a value and a pointer. The pointer contains the reference address to the next node in the list. If the value in the next node is NULL, then it is at the end of the list. As you can see, this type of list only allows you to go in one direction. You can start at the head of the list or if you have the reference address to an element in the list, you can access that element and iterate over the remaining elements, but you can't go backwards.

The LinkedList data structure in Java works the same way but it allows you to go forward and backward through a list of elements. That is why it is considered a doubly linked list. When using a LinkedList, each element stores a reference location for the next and previous element in the list. The first and last element are identified by looking at the reference value, the first element has a null pointer for the previous element, and the last element has a null pointer for the next element. There is also a reference value to the start of the list, referred to as the head of the list. One important difference about a LinkedList is that it does NOT use index values to reference objects. Here is a code snippet for creating a LinkedList and a diagram of the list that gets created:

```

List<String> ingredients = new LinkedList<>();
ingredients.add("apples");
ingredients.add("pie crust");
ingredients.add("sugar");

```

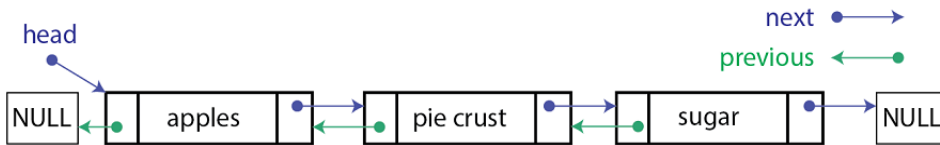


Figure 21.4 This diagram represents a doubly linkedlist in Java for apple pie ingredients

The next list type is called a Vector. A Vector can grow and shrink as needed, and it can be referenced using index values. The Vector Class is a carryover from older versions of Java. A key difference is that vectors are synchronized in Java. This means that they are thread safe, any method that uses a Vector will not allow access until the method returns control to the calling. A Vector is similar to an ArrayList, but it is considered thread safe. If your program is not using threads, this should not be a problem and you can use an ArrayList.

### Thread

A thread in Java is the term used to describe the path followed when executing a program. All Java programs have at least one thread, this is the main thread. The JVM creates the main thread when the program starts by finding the method declaration for the main method. Remember, every Java program MUST have one and only one main method. Java supports multi-threaded applications which is a powerful tool if your application needs to process large amounts of data concurrently.

Here is a code snippet for creating a Vector:

```
List<String> furniture = new Vector<>();
furniture.add("chair");
furniture.add("sofa");
furniture.add("end table");
```

In this code, I start by creating a new Vector of Strings. Next, I added three furniture items to the vector. Remember, vectors work very similar to an ArrayList, but they are synchronized. So, each application thread can only access the elements in the vector when all other threads have released the objects.

Consider a library. You can go to the library and check out a book. While you are reading it, no one else can check it out. When you are done, you return it to the library and now the next person can access the book. This is similar to the concept of synchronization in Java.

## 21.4 Common Collection Interface Methods

Since List, Set, and Queue all implement the Collection Interface, I wanted to provide a list of the methods in the

Collection Interface. These methods are automatically inherited by each of these sub-collections. Table 21.2 lists a subset of the methods that are included in the Collection Interface.

**Table 21.2 Collection interface methods**

| Method                                                  | Description                                                     |
|---------------------------------------------------------|-----------------------------------------------------------------|
| <code>boolean add(E e)</code>                           | returns true if element is successfully added                   |
| <code>boolean addAll(Collection&lt;?&gt; c)</code>      | returns true if all elements are added                          |
| <code>void clear()</code>                               | removes all elements from the collection                        |
| <code>boolean contains(Object o)</code>                 | returns true if this collection contains the specified element  |
| <code>boolean containsAll(Collection&lt;?&gt; c)</code> | returns true if all of the elements are in the collection       |
| <code>boolean equals(Object o)</code>                   | compares the specified object with the collection for equality  |
| <code>int hashCode()</code>                             | returns the hash code value for this collection                 |
| <code>boolean isEmpty()</code>                          | returns true if the collection has no elements                  |
| <code>Iterator&lt;E&gt; iterator()</code>               | returns an iterator that can be used to traverse all elements   |
| <code>boolean remove(Object o)</code>                   | returns true if element is found and removed from collection    |
| <code>boolean removeAll(Collection&lt;?&gt; c)</code>   | returns true if all elements are found and removed              |
| <code>boolean retainAll(Collection&lt;?&gt; c)</code>   | keeps only the elements in the specified collection             |
| <code>int size()</code>                                 | returns the number of elements in this collection               |
| <code>Object[] toArray()</code>                         | returns an array that includes all elements from the collection |

The methods listed in Table 21.2 are from the Collection Interface. In addition to this set of methods, each of the interfaces that implement Collection also have additional methods. For example, the List interface allows for adding an element to a certain index value: `boolean add(int index, E element)`.

#### Code Listing 21.2 Sample code to help demonstrate how to use List in Java

```

1. List<String> colors = new ArrayList<>();
2. List<Integer> grades = new LinkedList<>();
3. List<Character> alphabet = new Vector(26);
4.
5. List<String> rgb = new ArrayList<>();
6. rgb.add("red");
7. rgb.add("green");
8. rgb.add("blue");
9.
10. List<Integer> numbers = new LinkedList<>();
11. numbers.add(100);
12.

```



```

13. List<Character> letters = new Vector(26);
14. letters.add('b');

```

For each of the common methods, Table 21.3 shows a sample code snippet for an ArrayList, LinkedList and Vector. The examples use the code in Listing 21.2. For this example, each row uses the list from the previous row as a starting point.

**Table 21.3 Code samples for each of the common methods**

| Method                                            | Code snippet                                                                                                                          | Result                                                           |
|---------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------|
| <b>boolean add(E e)</b>                           | <code>colors.add("yellow");</code><br><code>grades.add(new Integer(95));</code><br><code>alphabet.add(new Character('a'));</code>     | [yellow]<br>[95]<br>[a]                                          |
| <b>boolean addAll(Collection&lt;?&gt; c)</b>      | <code>colors.addAll(rgb);</code><br><code>grades.add(numbers);</code><br><code>alphabet.add(letters);</code>                          | [yellow, red, green, blue]<br>[95, 100]<br>[a, b]                |
| <b>boolean contains(Object o)</b>                 | <code>colors.contains("green");</code><br><code>grades.contains(85);</code><br><code>alphabet.contains('b');</code>                   | true<br>false<br>true                                            |
| <b>boolean containsAll(Collection&lt;?&gt; c)</b> | <code>colors.containsAll(rgb);</code><br><code>grades.containsAll(numbers);</code><br><code>alphabet.containsAll(letters);</code>     | true<br>true<br>true                                             |
| <b>boolean equals(Object o)</b>                   | <code>colors.equals(rgb);</code><br><code>grades.equals(numbers);</code><br><code>alphabet.equals(letters);</code>                    | false<br>false<br>false                                          |
| <b>int hashCode()</b>                             | <code>int hc1 = colors.hashCode();</code><br><code>int hc2 = grades.hashCode();</code><br><code>int hc3 = alphabet.hashCode();</code> | each integer now contains a numeric has code, i.e.:<br>205596058 |
| <b>boolean isEmpty()</b>                          | <code>colors.isEmpty();</code><br><code>grades.isEmpty();</code><br><code>alphabet.isEmpty();</code>                                  | false<br>false<br>false                                          |
| <b>boolean remove(Object o)</b>                   | <code>colors.remove("green");</code><br><code>grades.remove(87);</code><br><code>alphabet.remove('a');</code>                         | [yellow, red, blue]<br>[95, 100]<br>[b]                          |
| <b>boolean removeAll()</b>                        | <code>colors.removeAll(rgb);</code>                                                                                                   | [yellow]                                                         |

|                                                            |                                                                                                                             |                                                                           |
|------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------|
| <code>removeAll(Collection&lt;?&gt; c)</code>              | <code>grades.removeAll(numbers);</code><br><code>alphabet.removeAll(letters);</code>                                        | [95]<br>[b]                                                               |
| <code>boolean<br/>retainAll(Collection&lt;?&gt; c)*</code> | <code>colors.retainAll(rgb);</code><br><code>grades.retainAll(numbers);</code><br><code>alphabet.retainAll(letters);</code> | []<br>[]<br>[b]                                                           |
| <code>int size()</code>                                    | <code>colors.size();</code><br><code>grades.size();</code><br><code>alphabet.size();</code>                                 | returns the size of the list, which is the number of elements in the list |
| <code>Object[] toArray()</code>                            | <code>colors.toArray();</code><br><code>grades.toArray();</code><br><code>alphabet.toArray();</code>                        | converts the list to an array                                             |
| <code>void clear()</code>                                  | <code>colors.clear();</code><br><code>grades.clear();</code><br><code>alphabet.clear();</code>                              | []<br>[]<br>[]                                                            |
| <code>Iterator&lt;E&gt; iterator()</code>                  | <code>colors.iterator();</code><br><code>grades.iterator();</code><br><code>alphabet.iterator();</code>                     | returns an Iterator that can be used to traverse the list                 |

\* In the table above, I wanted to take a closer look at the `retainAll` method. This method is similar to the intersection of two lists. So, only elements that occur in both lists are returned from this method.

The table starts with the `ArrayList`, `LinkedList`, and `Vector` from listing 21.2. To help demonstrate the methods, I made the `ArrayList` a list of `String` objects, the `LinkedList` contains `Integer` values, and the `Vector` contains `Character` values. The third column shows the result of the list after each method statement is executed.

### Quick Check 21-3:

Given this code listing, answer the questions below:

#### Listing 21.3 Sample code for Quick Check 21-3

```

1. package lesson21_collection;
2. import java.util.ArrayList;
3. import java.util.LinkedList;
4. import java.util.List;
5. import java.util.Vector;
6.
7. public class Lesson21_List {
8. public static void main(String[] args) {
9. List<String> dogs = new ArrayList<>();
10. dogs.add("Whippet");
11. dogs.add("Beagle");

```

```

12. dogs.add("Golden Retriever");
13. dogs.add(0,"Labrador Retriever");
14. System.out.println(dogs);
15.
16. List<String> movies = new LinkedList<>();
17. movies.add("Doctor Strange");
18. movies.add(1,"Avengers");
19. movies.add("Black Panther");
20. System.out.println(movies);
21. }
22. }

```

1. What is printed after executing line 14:
  - a. [Whippet, Beagle, Golden Retriever, Labrador Retriever]
  - b. [Labrador Retriever, Whippet, Beagle, Golden Retriever]
  - c. [Whippet, Labrador Retriever, Beagle, Golden Retriever]
  - d. [Labrador Retriever, Golden Retriever, Beagle, Whippet]
2. What is printed after executing line 26:
  - a. [Doctor Strange, Avengers, Black Panther]
  - b. [Avengers, Doctor Strange, Black Panther]
  - c. [Avengers, Black Panther, Doctor Strange]
  - d. [Black Panther, Avengers, Doctor Strange]

## 21.5 Summary

In this lesson, you learned:

- What interfaces and classes are included in the Collections Framework
- When to use the List Interface and the classes that extend that interface
- How to create lists using the classes ArrayList, LinkedList, and Vector
- About the common collection interface methods that are available for the List collection

This lesson started by introducing the Collection framework in Java. The framework has various collections that can be used to store, retrieve and manipulate data. Choosing the right collection is important to ensure your application runs efficiently and securely. This lesson reviewed the criteria that results in the creation of a List data structure. Next, the lesson reviewed the process to create each type of list, including ArrayList, LinkedList, and Vector.

The next lesson continues the discussion of Collections and reviews the Set Interface and the classes that extend Set.

### **Try this:**

Have you ever created a playlist of songs? How do you listen to the songs on the playlist? I'm guessing that sometimes you listen to each song sequentially or even in reverse. You might have a favorite song in the middle of the list that you listen to frequently. Looking at our

decision chart, follow the path to choose a list type and write the code for creating the list. There are several possible options, so choose one and then create the list.

### Quick Check 21-1:

1. Identify the interface(s) that are implemented by the `LinkedList` Class:

- a. **List**
- b. Set
- c. **Queue**
- d. Map

2. Which class(es) can be used to complete this code snippet:

```
List list1 = new _____ <String>();
```

- a. List
- b. **Vector**
- c. HashSet
- d. **LinkedList**

3. Which class(es) can be used to complete this code snippet:

```
Map<Integer, String[]> map1 = new _____ <Integer, String[]>();
```

- a. Map
- b. SortedMap
- c. **HashMap**
- d. **LinkedHashMap**

### Quick Check 21-2 Solution:

**Use the decision tree to find the appropriate choice for these scenarios:**

1. You have a collection of elements that need to be sorted in their natural order, and each element has a unique string associated with its value, which is best suited for your needs?

- a. ArrayList
- b. HashMap
- c. HashSet
- d. **TreeMap**

### Quick Check 21-3 Solution:

Given this code listing, answer the questions below:

#### Listing 21.3 Sample code for Quick Check 21-3

```
1. package lesson21_collection;
```

```

2. import java.util.ArrayList;
3. import java.util.LinkedList;
4. import java.util.List;
5. import java.util.Vector;
6.
7. public class Lesson21_List {
8. public static void main(String[] args) {
9. List<String> dogs = new ArrayList<>();
10. dogs.add("Whippet");
11. dogs.add("Beagle");
12. dogs.add("Golden Retriever");
13. dogs.add(0,"Labrador Retriever");
14. System.out.println(dogs);
15.
16. List<String> movies = new LinkedList<>();
17. movies.add("Doctor Strange");
18. movies.add(1,"Avengers");
19. movies.add("Black Panther");
20. System.out.println(movies);
21. }
22. }

```

1. What is printed after executing line 14:
  - a. [Whippet, Beagle, Golden Retriever, Labrador Retriever]
  - b. [Labrador Retriever, Whippet, Beagle, Golden Retriever]**
  - c. [Whippet, Labrador Retriever, Beagle, Golden Retriever]
  - d. [Labrador Retriever, Golden Retriever, Beagle, Whippet]
  
2. What is printed after executing line 26:
  - a. [Doctor Strange, Avengers, Black Panther]**
  - b. [Avengers, Doctor Strange, Black Panther]
  - c. [Avengers, Black Panther, Doctor Strange]
  - d. [Black Panther, Avengers, Doctor Strange]

### **Solution to the Try This activity:**

Using the decision chart, I decided to use a LinkedList. A list of songs in a playlist might have duplicates, most of the time I want to access values at specific positions in the list, but I frequently start at the beginning or end, and the songs on the playlist might change frequently.

#### **Listing 21.4 Code to create a song playlist**

```

1. package lesson21_collection;
2.
3. import java.util.LinkedList;
4. import java.util.List;
5.
6. public class Lesson21_Playlist {
7. public static void main(String[] args) {
8. List<String> playlist = new LinkedList<>();

```

```
9. playlist.add("Don't Stop Believin");
10. playlist.add("Faithfully");
11. playlist.add("Any Way You Want It");
12. }
13. }
```

# 22

## Java Collections: Set

**After reading this lesson, you will be able to:**

- Understand when to use the Set collection
- Create sets using the HashSet, LinkedHashSet, and TreeSet classes
- Apply methods available for the Set collection in a Java program

This lesson introduces the types of Set collections Java. It includes the benefits of using and the difference between the types of Sets. A Set is an object that represents another group of objects that does not allow for any duplicates.

### Consider This

A local zoo is merging with another zoo and they want to create a brochure with the complete list of the types of animals that will now be in the new zoo. For this program, then want to enter the types of animals from each zoo, but they don't want duplicates. They also want to have access to the animal names alphabetically to print on the brochure. This scenario is perfect for using a TreeSet, since that does not allow for duplicates and it requires the items to be sorted.

### 22.1 Sets in Java

In this lesson, we are concentrating on the Set interface. Lesson 21 introduces the Java Collection along with an overview of all the interfaces and classes that extend from the Collection Interface. Figure 22.1 shows the Java Collection Framework with only the Set interfaces and classes. As you can see, there are two classes that extend directly from the Set interface: HashSet and LinkedHashSet. The TreeSet extends the SortedSet interface which extends the Set interface. The TreeSet is different since it requires the elements to be sorted, but it still follows the rule that all elements in a Set must be unique, no duplicates.

## Java Collection Framework

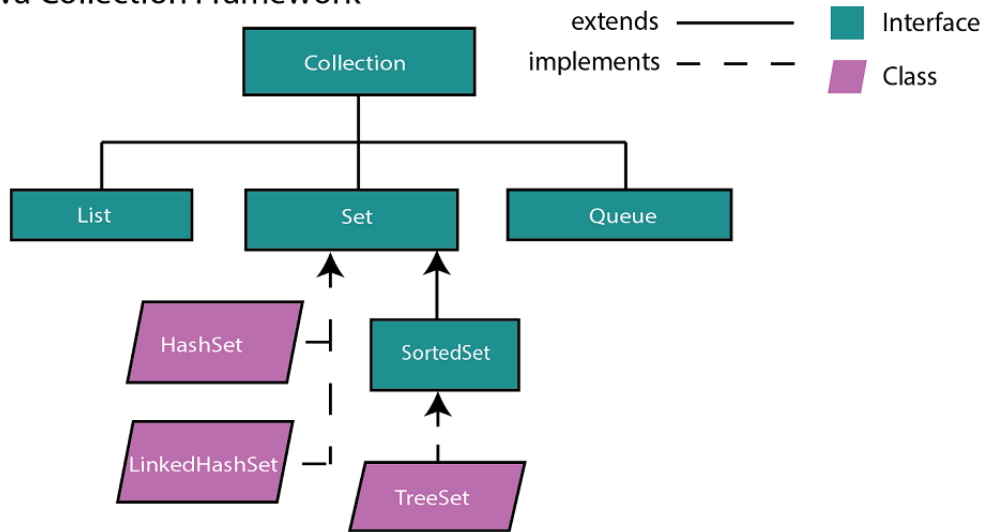


Figure 22.1 Diagram of the Set portion of the Java Collection Framework

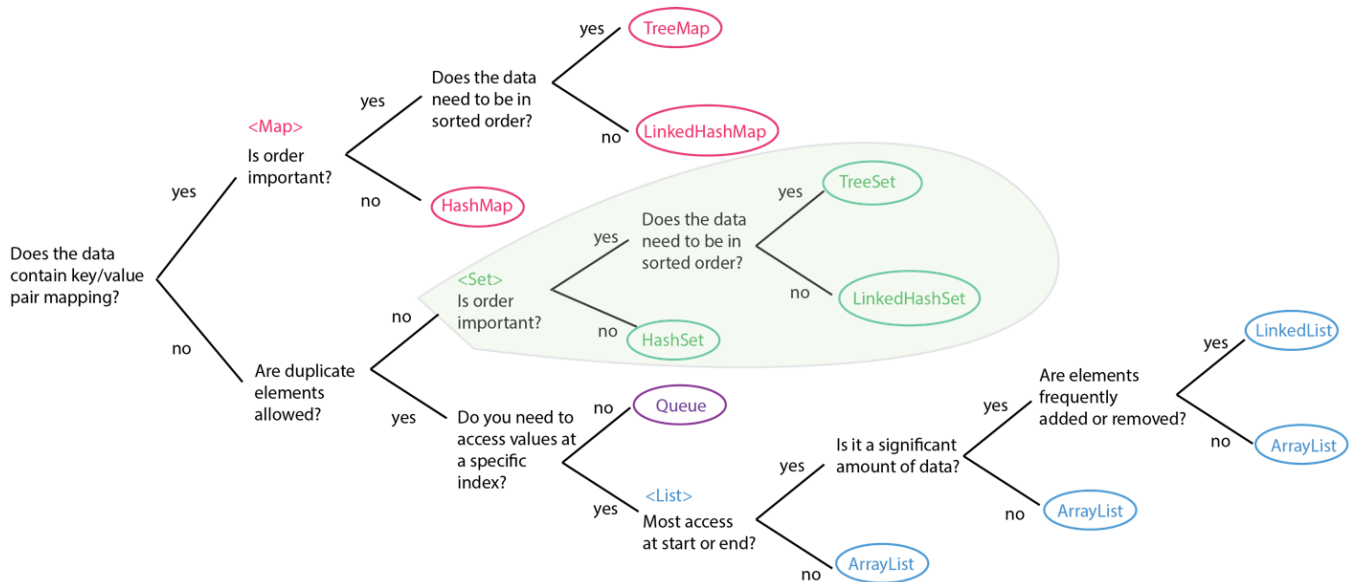
### Quick Check 22-1:

1. Which class is best suited to store a sorted list of elements with no duplicates:
  - a. HashSet
  - b. Set
  - c. TreeSet
  - d. LinkedHashSet

## 22.2 Understand when to use the Set collection

Choosing the right Collection can make a big difference in your application performance, readability, and reuse. To help decide which Collection to use, I've created a decision tree as seen in Figure 22.5. This time the highlighted section identifies the criteria for using the Set interface.





**Figure 22.2** A decision tree for choosing the right Collection in Java, notice the highlighted section for Set types

The highlighted portion of the decision tree indicates the questions used to choose a Set type. Follow each question by answering yes/no until you reach the appropriate collection type. For example, in our Consider This section, we start with the Set interface since there can be no duplicates. Next, is the order important? In our example it is important since we want the animals to be in alphabetical order. The last question is whether or not it needs to be sorted, and we answered yes, so we would use a TreeSet.

These are not strict rules that have to be followed, but this helps to choose the most efficient collection type for your application.

### Quick Check 22-2:

Use the decision tree to find the appropriate choice for these scenarios:

1. You want to store a collection of books (which can contain duplicates) and you want to have access to print them out based on a position in the list, which is best suited for your needs?
  - a. ArrayList
  - b. HashMap
  - c. HashSet
  - d. TreeMap

## 22.3 HashSet, LinkedHashSet, and TreeSet

One important thing to remember when working with collections, is that you cannot instantiate an interface.

Table 22.1 provides information about each of the Set classes: HashSet, LinkedHashSet, and TreeSet. One common attribute to all Set collections is that they do not allow for duplicate values.

**Table 22.1 Shows each type of Set and a brief description of their differences**

| Set Type      | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| HashSet       | <p>A list of elements that are stored using a hash table</p> <p>A hash code is calculated for each element and that value is used to store the value in the table using the hash code as the key or index to that value</p> <p>It allows for null values</p> <p>Because it uses the hash code to store the elements, it does not maintain the insertion order</p> <p>It is a good choice if you need to frequently search for an element</p> <p>The initial size is 16 with a load factor of .75, which means that the size will be increased once the set is at 75% capacity</p> |
| LinkedHashSet | <p>Combines a hash set with a linked list, elements are stored using a hash code, and then added to a linked list</p> <p>By using a linked list to store the elements, it maintains the insertion order which makes it different from the HashSet</p> <p>It allows for null values</p>                                                                                                                                                                                                                                                                                            |
| TreeSet       | <p>Access and retrieval from a TreeSet are fast</p> <p>Maintains an ascending order of the elements</p> <p>Does NOT allow for null values</p>                                                                                                                                                                                                                                                                                                                                                                                                                                     |

If you have a list of elements and you only want to identify the unique items in the list, you can add the list collection to any of the Set types listed above and only one copy of each unique element will be added. Code Listing 22.1 shows an example of using a Set to create a unique list of elements.

### Code Listing 22.1 sample code to create a set from a list

```

1. package lesson22;
2.
3. import java.util.ArrayList;
4. import java.util.Arrays;
5. import java.util.HashSet;
6. import java.util.List;
7. import java.util.Set;

```

```

8.
9. public class Lesson22_Listing1 {
10. public static void main(String[] args) {
11. List customers = new ArrayList();
12. customers.add("customer1");
13. customers.add("customer2");
14. customers.add("customer3");
15. customers.add("customer1");
16. Set uniqueCustomers = new HashSet(); // #A
17. uniqueCustomers.addAll(customers); // #B
18. System.out.println(customers.toString());
19. System.out.println(Arrays.toString(uniqueCustomers.toArray()));
20. }
21. }

```

#A Create a new HashSet

#B Add the list of customers to the new HashSet which automatically ignores duplicates

In this example, I have started by creating a list of four customers, including one duplicate customer. Then I created a HashSet and added the list of customers to that set. This allows the code to automatically remove any duplicates. The output from this code snippet shows the original list and then prints the elements in the HashSet:

```

[customer1, customer2, customer3, customer1]
[customer2, customer3, customer1]

```

As you can see, the second line of output only contains one element for customer1. It is also noticeable that the HashSet does not retain the insertion order of the elements. The next three sections take a closer look at each type of Set.

### 22.3.1 HashSet

We can create any of these set types with a variable declared as a Set reference type since it is the super class for each of these subclasses. Listing 22.2 creates a Set reference variable, numbers, that is instantiated as a HashSet. Next, the for loop adds the numbers 1<sup>2</sup> through 10<sup>2</sup>. The output from this code might look like this: [16, 64, 1, 49, 81, 4, 36, 100, 9, 25].

#### Code Listing 22.2 sample code to create a set of values in a HashSet

```

1. package lesson22;
2.
3. import java.util.Arrays;
4. import java.util.HashSet;
5. import java.util.Set;
6.
7. public class Lesson22_Listing2 {
8.
9. public static void main(String[] args) {
10. Set numbers = new HashSet(); // #A
11. for (int i = 1; i <= 10; i++) { // #B
12. numbers.add(i * i);
13. }
14. System.out.println(numbers.contains(100)); // #C

```

```

15. System.out.println(Arrays.toString(numbers.toArray()));
16. }
17. }

```

#A This statement creates a Set reference variable to a HashSet

#B Add the values  $1^2$ -  $10^2$  to the HashSet

#C Use the contains method to check for the value 100

A HashSet does not guarantee any particular order.

Figure 22.3 shows how the set might look for this example:

HashSet: [16, 64, 1, 49, 81, 4, 36, 100, 9, 25]

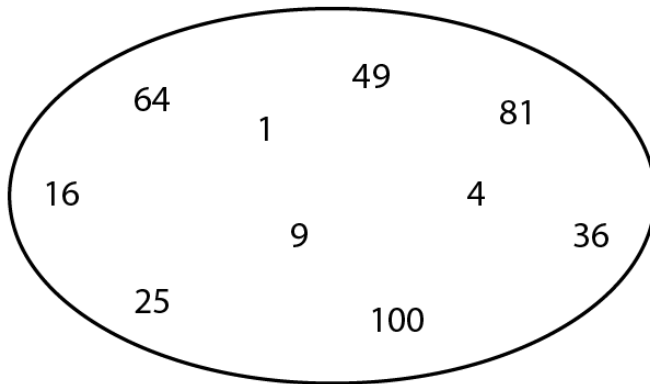


Figure 22.3 shows an example of a HashSet containing the values from the numbers set in Listing 22.2

The HashSet makes searching for an element quick, line 14 searches the collection for a specific value, in this case 100, which returns true.

### 22.3.2 LinkedHashMap

The difference between a LinkedHashMap and a HashSet is that it allows us to store the elements in the order they are added to the set. The LinkedHashMap data structure in Java works similar to the LinkedList from Lesson 21. It allows you to go forward and backward through a set of elements. Here is a code snippet that uses a LinkedHashMap to store the same values from the HashSet example:  $1^2$  through  $10^2$ .

Listing 22.3 creates a Set reference variable, numbers, that is instantiated as a LinkedHashMap. This time the output is in insertion order producing a different output than the HashSet: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100].

**Code Listing 22.3 sample code to create a set of values using a HashSet**

```

1. package lesson22;
2.
3. import java.util.Arrays;
4. import java.util.HashSet;
5. import java.util.Set;
6.
7. public class Lesson22_Listing3 {
8.
9. public static void main(String[] args) {
10. Set numbers = new HashSet(); //A
11. for (int i = 1; i <= 10; i++) {
12. numbers.add(i * i);
13.
14. System.out.println(Arrays.toString(numbers.toArray())); //B
15. }
16. }

```

#A Create a Set reference variable to a HashSet

#B Prints the set by first converting it to an Array and then using the static class Arrays to print the values

The linked list shown in figure 22.4 shows how the elements are inserted into the set based on insertion order. Since the set is a linked set, it follows the same rules as the LinkedList, creating a double linked set of elements.

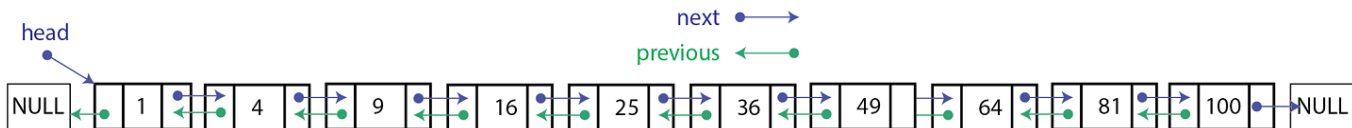


Figure 22.4 This figure shows the HashSet created by the code in Listing 22.3

### 22.3.3 TreeSet

Code listing 22.4 uses a TreeSet to hold a list of animals at the zoo:

**Code Listing 22.4 sample code to create a set of values in sorted order using a TreeSet**

```

1. package lesson22;
2.
3. import java.util.Arrays;
4. import java.util.Set;
5. import java.util.TreeSet;
6.
7. public class Lesson22_Listing4 {
8. public static void main(String[] args){
9. Set animals = new TreeSet(); //A
10. animals.add("Lion"); //B
11. animals.add("Bear"); //B
12. animals.add("Tiger"); //B
13. System.out.println(Arrays.toString(animals.toArray()));
14. }

```

```
15. }
```

**#A Create a Set reference variable to a TreeSet**

**#B Add random items to the TreeSet which stores the elements in sorted order**

Listing 22.4 creates a Set reference variable, `animals`, that is instantiated as a `TreeSet`. Three animals are added to the set. Since the `TreeSet` stores the elements in sorted order, the output is: `[Bear, Lion, Tiger]`. The `TreeSet` collection can be created using the `Set` interface, the `SortedSet` interface and the class name `TreeSet`. Listing 22.5 is similar to the code listing from 22.4, but this time there are two differences. On line 9 I have changed the set type from `Set` to `SortedSet` and added an additional statement on line 13.

#### Code Listing 22.5 sample code to create a set of values in sorted order using a TreeSet

```
1. package lesson22;
2.
3. import java.util.Arrays;
4. import java.util.SortedSet;
5. import java.util.TreeSet;
6.
7. public class Lesson22_Listing5 {
8. public static void main(String[] args){
9. SortedSet animals = new TreeSet();
10. animals.add("Lion");
11. animals.add("Bear");
12. animals.add("Tiger");
13. animals.add("Bear"); // #A
14. System.out.println(Arrays.toString(animals.toArray()));
15. }
16. }
```

**#A Add the element "Bear" twice, but the output shows that it is only stored once in the set**

In Listing 22.5, since the `animals` set does not allow duplicates, the statement on line 13 to add `Bear` to the set a second time is ignored and this code produces the exact same result: `[Bear, Lion, Tiger]`. When adding a duplicate element, it does not throw an exception error, instead it returns `true` if the element is added and `false` if the element already exists and was not added a second time.

#### Quick Check 22-3:

Given this code listing, answer the questions below:

#### Code Listing 22.6 sample code for the quick check 22-3

```
1. package lesson22;
2.
3. import java.util.Arrays;
4. import java.util.Set;
5. import java.util.HashSet;
6. import java.util.LinkedHashSet;
7. import java.util.TreeSet;
```

```

8.
9. public class Lesson22_Listing6 {
10. public static void main(String[] args){
11. Set set1 = new HashSet();
12. Set set2 = new LinkedHashSet();
13. Set set3 = new TreeSet();
14. for(int i = 1; i <= 10; i++) {
15. set1.add(i*i);
16. set2.add(i*i);
17. set3.add(i*i);
18. }
19. set1.add(51);
20. set2.add(2);
21. set3.add(47);
22. }
23. }

```

1. What is the output from this statement:

```
System.out.println(Arrays.toString(set1.toArray()));
```

- a. [16, 64, 1, 49, 81, 51, 4, 36, 100, 9, 25]
- b. [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 2]
- c. [1, 4, 9, 16, 25, 36, 47, 49, 64, 81, 100]
- d. The output contains all values listed above, but it is not possible to determine the order

2. What is the output from this statement:

```
System.out.println(Arrays.toString(set2.toArray()));
```

- a. [16, 64, 1, 49, 81, 51, 4, 36, 100, 9, 25]
- b. [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 2]
- c. [1, 4, 9, 16, 25, 36, 47, 49, 64, 81, 100]
- d. The output contains all values listed above, but it is not possible to determine the order

3. What is the output from this statement:

```
System.out.println(Arrays.toString(set3.toArray()));
```

- a. [16, 64, 1, 49, 81, 51, 4, 36, 100, 9, 25]
- b. [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 2]
- c. [1, 4, 9, 16, 25, 36, 47, 49, 64, 81, 100]
- d. The output contains all values listed above, but it is not possible to determine the order

4. Your application is designed to process a series of invoices from various customers. A single customer might have more than one invoice with a unique invoice number. You need to create a mailing list of the customers contained in the list of invoices. Which type of Set collection should you use to create the set of unique customer names in alphabetical order?

- a. HashSet
- b. LinkedHashSet

- c. TreeSet
- d. None of the above

## 22.4 Common Collection Interface Methods

Since Set, HashSet, LinkedHashSet, and TreeSet all implement the Collection Interface, I wanted to provide a reference to the common methods in the Collection Interface. These methods are automatically inherited by each of these sub-collections. Table 22.2 lists a subset of the methods that are included in the Collection Interface.

In Table 22.2, I wanted to take a closer look at the retainAll method. This method is similar to the intersection of two sets. So, only elements that occur in both sets are returned from this method.

**Table 22.2** Collection interface methods

| Method                               | Description                                                      |
|--------------------------------------|------------------------------------------------------------------|
| boolean add(E e)                     | returns true if element is successfully added                    |
| boolean addAll(Collection<?> c)      | returns true if all elements are added                           |
| void clear()                         | removes all elements from the collection                         |
| boolean contains(Object o)           | returns true if this collection contains the specified element   |
| boolean containsAll(Collection<?> c) | returns true if all of the elements are in the collection        |
| boolean equals(Object o)             | compares the specified object with the collection for equality   |
| int hashCode()                       | returns the hash code value for this collection                  |
| boolean isEmpty()                    | returns true if the collection has no elements                   |
| Iterator<E> iterator()               | returns an iterator that can be used to traverse all elements    |
| boolean remove(Object o)             | returns true if element is found and removed from collection     |
| boolean removeAll(Collection<?> c)   | returns true if all elements are found and removed               |
| boolean retainAll(Collection<?> c)   | keeps only the elements in the specified collection (*see below) |
| int size()                           | returns the number of elements in this collection                |
| Object[] toArray()                   | returns an array that includes all elements from the collection  |

The methods listed in Table 22.2 are from the Collection Interface. In addition to these methods, each of the interfaces that implement Collection also have additional methods. For example, the SortedSet interface contains additional methods, for example: subset, headSet and tailSet, first, and last. You can find all the methods for each class in the Oracle JavaDocs: <https://docs.oracle.com/en/java/javase>



Table 22.3 shows a subset of unique methods for the SortedSet.

**Table 22.3 A subset of some unique methods for the SortedSet Interface**

| Method                                          | Description                                                                                 |
|-------------------------------------------------|---------------------------------------------------------------------------------------------|
| SortedSet<E> subSet(E fromElement, E toElement) | returns all elements from the first element (inclusive) up to the toElement (not inclusive) |
| SortedSet<E> headSet(E toElement)               | returns all elements up to the toElement (not inclusive)                                    |
| SortedSet<E> tailSet(E fromElement)             | returns all elements after to fromElement to the end (inclusive)                            |
| E first()                                       | returns the first element in the set                                                        |
| E last()                                        | returns the last element in the set                                                         |

To demonstrate how to use these methods, I have created a code example, shown here in Listing 22.7, that demonstrates how to use these additional methods for a SortedSet.

The output from code listing 22.7 is shown below in figure 22.5. Make sure you look closely at the output, it might be different than what you expect.

```
[Blue, Green, Indigo, Orange, Red, Violet, Yellow]
First element: Blue
Last element: Yellow
Subset: [Indigo, Orange, Red, Violet]
Head of the set: []
Tail element: [Yellow]
```

**Figure 22.5 Snapshot of the output from executing the code snippet in Listing 22.7**

#### Listing 22.7 Sample code for the additional SortedSet methods

```
1. package lesson22;
2. import java.util.Arrays;
3. import java.util.SortedSet;
4. import java.util.TreeSet;
5.
6. public class Lesson22_Listing7 {
7. public static void main(String[] args){
8. SortedSet set1 = new TreeSet();
9. set1.add("Red");
10. set1.add("Orange");
11. set1.add("Yellow");
12. set1.add("Green");
13. set1.add("Blue");
14. set1.add("Indigo");
15. set1.add("Violet");
16. System.out.println(Arrays.toString(set1.toArray()));
17. System.out.println("First element: " + set1.first()); //A
```

```

18. System.out.println("Last element: " + set1.last()); //B
19. System.out.println("Subset: " + set1.subSet("Indigo", "Yellow")); //C
20. System.out.println("Head of the set: " + set1.headSet("Blue")); //D
21. System.out.println("Tail element: " + set1.tailSet("Yellow")); //E
22. }
23. }

```

**#A** Use the first method from the SortedSet to print the first element from the set

**#B** Print the last element from the set

**#C** Print a subset of the rainbow elements from Indigo, not including Yellow

**#D** Print the front part of the set up to the element "Blue" but not including Blue

**#E** Print the last part of the set from Yellow to the end

At first glance the output looks incorrect but remember that a TreeSet adds the elements in sorted order to the set, it does not maintain the insertion order. So, in this example, the first element is not Red, instead it is the first color alphabetically which is Blue.

### Quick Check 22-4:

Given this code listing, answer the questions below:

#### Listing 22.8 Sample code for Quick Check 22-4

```

1. package lesson22;
2.
3. import java.util.Arrays;
4. import java.util.HashSet;
5. import java.util.LinkedHashSet;
6. import java.util.Set;
7. import java.util.SortedSet;
8. import java.util.TreeSet;
9.
10. public class Lesson22_Listing8 {
11. public static void main(String[] args){
12. SortedSet set1 = new TreeSet();
13. set1.add("one");
14. set1.add("two");
15. set1.add("three");
16. Set set2 = new HashSet();
17. Set set3 = new LinkedHashSet();
18. //A
19. set3 = set2;
20. System.out.println(set3.contains("three")); //B
21. //C
22. System.out.println(set3.size()); //D
23. }
24. }

```

1. Which statement replaces //A to add the elements in set1 to set2:

- a. `set2.add(set1);`
- b. `set2.addAll(set1);`
- c. `set2.contains(set1);`
- d. `set1.addAll(set2);`

2. What is printed after executing the statement at //B:
  - a. true
  - b. false
  - c. 0
  - d. null pointer exception
3. Which statement can replace //C to add an element to set3:
  - a. `set3.add("four");`
  - b. `set3.addAll("four");`
  - c. `set3.put("four");`
  - d. null pointer exception
4. What is printed after executing the statement at //D:
  - a. 3
  - b. 0
  - c. 4
  - d. null pointer exception

This lesson started by reviewing the decision tree to identify scenarios that are best suited by a Set collection. All Set collections ignore duplicate elements. Each set type is unique, the HashSet stores elements using a hash code and does not maintain insertion order. The LinkedHashSet is similar to the HashSet, but it maintains the insertion order. Finally, the TreeSet is unique since the elements are stored in sorted order and it provides some unique methods based on the fact that the elements are sorted.

The next lesson continues the discussion of Collections and reviews the Queue Interface and the classes that extend Queue.

## 22.5 Summary

In this lesson, you learned:

- How to identify situations that work best by using a Set to store the elements
- To create sets using the HashSet, LinkedHashSet, and TreeSet classes and add elements to each
- How to apply methods available for the Set collection in a Java program

### **Try this:**

You need to write a program that maintains the insertion order, but also automatically eliminates any duplicates. Write a sample program that stores a list of items, including some duplicates, and then print the set of elements.

**Answer Section:****Quick Check 22-1 Solution:**

1. Which class is best suited to store a sorted list of elements with no duplicates:
  - a. HashSet
  - b. Set
  - c. **TreeSet**
  - d. LinkedHashSet

**Quick Check 22-2 Solution:****Use the decision tree to find the appropriate choice for these scenarios:**

1. You want to store a collection of books (which can contain duplicates) and you want to have access to print them out based on a position in the list, which is best suited for your needs?
  - a. **ArrayList**
  - b. HashMap
  - c. HashSet
  - d. TreeMap

**Quick Check 22-3 Solution:****Given this code listing, answer the questions below:****Code Listing 22.5 sample code for the quick check 22-3**

```

1. package lesson22;
2.
3. import java.util.Arrays;
4. import java.util.Set;
5. import java.util.HashSet;
6. import java.util.LinkedHashSet;
7. import java.util.TreeSet;
8.
9. public class Lesson22_Listing5 {
10. public static void main(String[] args){
11. Set set1 = new HashSet();
12. Set set2 = new LinkedHashSet();
13. Set set3 = new TreeSet();
14. for(int i = 1; i <= 10; i++) {
15. set1.add(i*i);
16. set2.add(i*i);
17. set3.add(i*i);
18. }
19. set1.add(51);
20. set2.add(51);
21. set3.add(51);
22. }
23. }

```

1. What is the output from this statement: `System.out.println(Arrays.toString(set1.toArray()));`

- a. [16, 64, 1, 49, 81, 51, 4, 36, 100, 9, 25]  
 b. [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 2]  
 c. [1, 4, 9, 16, 25, 36, 47, 49, 64, 81, 100]  
**d. The output contains all values listed above, but it is not possible to determine the order**
2. What is the output from this statement: `System.out.println(Arrays.toString(set2.toArray()));`
- a. [16, 64, 1, 49, 81, 51, 4, 36, 100, 9, 25]  
**b. [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 2]**  
 c. [1, 4, 9, 16, 25, 36, 47, 49, 64, 81, 100]  
 d. The output contains all values listed above, but it is not possible to determine the order
3. What is the output from this statement: `System.out.println(Arrays.toString(set3.toArray()));`
- a. [16, 64, 1, 49, 81, 51, 4, 36, 100, 9, 25]  
 b. [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 2]  
**c. [1, 4, 9, 16, 25, 36, 47, 49, 64, 81, 100]**  
 d. The output contains all values listed above, but it is not possible to determine the order
4. Your application is designed to process a series of invoices from various customers. A single customer might have more than one invoice with a unique invoice number. You need to create a mailing list of the customers contained in the list of invoices. Which type of Set collection should you use to create the set of unique customer names in alphabetical order?
- a. HashSet  
 b. LinkedHashSet  
**c. TreeSet**  
 d. None of the above

### Quick Check 22-4 Solutions:

Given this code listing, answer the questions below:

#### Listing 22.8 Sample code for Quick Check 22-4

```

1. package lesson22;
2.
3. import java.util.Arrays;
4. import java.util.HashSet;
5. import java.util.LinkedHashSet;
6. import java.util.Set;
7. import java.util.SortedSet;
8. import java.util.TreeSet;
9.
10. public class Lesson22_Listing8 {
11. public static void main(String[] args){
12. SortedSet set1 = new TreeSet();
13. set1.add("one");
14. set1.add("two");
15. set1.add("three");
16. Set set2 = new HashSet();

```

```

17. Set set3 = new LinkedHashSet();
18. //A
19. set3 = set2;
20. System.out.println(set3.contains("three")); //B
21. //C
22. System.out.println(set3.size()); //D
23. }
24. }
25.

```

1. Which statement replaces //A to add the elements in set1 to set2:
  - a. set2.add(set1);
  - b. set2.addAll(set1);**
  - c. set2.contains(set1);
  - d. set2.containsAll(set1);
2. What is printed after executing the statement at //B:
  - a. true**
  - b. false
  - c. 0
  - d. null pointer exception
3. Which statement can replace //C to add an element to set3:
  - a. set3.add("four");**
  - b. set3.addAll("four");
  - c. set3.put("four");
  - d. null pointer exception
4. What is printed after executing the statement at //D:
  - a. 3
  - b. 0
  - c. 4**
  - d. null pointer exception

### ***Solution to the Try This activity:***

#### **Listing 22.9 Solution for Try This**

```

1. package lesson22;
2.
3. import java.util.Arrays;
4. import java.util.LinkedHashSet;
5. import java.util.Set;
6.
7. public class Lesson22_Listing9 {
8.
9. public static void main(String[] args) {
10. Set names = new LinkedHashSet(); //A

```

```
11. names.add("Liam");
12. names.add("Aurora");
13. names.add("Owen");
14. names.add("Samuel");
15. names.add("Lily");
16. names.add("Charlotte");
17. names.add("Jasper");
18. names.add("Aurora"); //B
19. names.add("Owen"); //B
20. System.out.println(Arrays.toString(names.toArray()));
21. }
22. }
```

**#A** Create a `LinkedHashSet` to maintain the insertion order

**#B** Add duplicates to the Set, when the set is printed, the duplicates are not there

Sample output: [Liam, Aurora, Owen, Samuel, Lily, Charlotte, Jasper]