Object Oriented Programming in Java

Attend class lecturers from home

VASKARAN SARCAR

Object Oriented Programming in Java

Attend class lecturers from home

By Vaskaran Sarcar

ME, MCA, B.Sc. (Math)

[Version: 1.0]

License Notes

A letter from Author

Dear Reader ,

Welcome to the journey. It is my privilege to enclose herewith *Object Oriented Programming in Java: Attend class lecturers from home*. Before jump into the topic, I want to highlight few points about the topic and its contents:

1. The aim of this book is to help you to get a feel of a Java classroom environment. I was involved in teaching since 2005.I have taken classes in both engineering and non-engineering colleges. And fortunately most of my teaching involvement was based on Java and its advanced topics. That is the true motivation to introduce a book like this.

2. *This book will not discuss how to write an if-else statement or a simple while loop etc. Your teacher expects that before attending the class, you have done your basic homework. Here your teacher will focus on the basic object oriented concepts that we can implement in Java.*

3. With this book you will have a feel that you are learning Java in a classroom environment-where your teacher will discuss about some problems/topics, ask you questions and give you assignments. You will be encouraged to do those simple assignments before entering into a new topic. If you are dedicated to this subject and do those assignments, you will surely develop the confidence on this language.

4. In a semester, there is a certain number of lectures to complete the fundamental topics. And we all know that learning is a continuous process. *So, this book is not for those who want to learn Java in 24 hours or in 7 days*. It is up to you only. I can only say : the book is designed for you in such a way that after completion of the book, you will develop an adequate knowledge on the topic, you'll learn the key features of this powerful language, you'll learn how we should write programs in Java and most importantly, how to go further.

5. Lastly the programs are tested with eclipse. Though it is not mandatory for you to learn eclipse. You can simply run these programs in your preferred IDE. Author has chosen eclipse because it is widely used to develop Java applications.

The Author

Contents

Basic Terms

JVM

-It stands for Java Virtual Machine. When we compile the java file, we get a .class (not an .exe).This file contains java byte code which is interpreted by JVM. It is responsible for loading, verifying and executing the code .We say that JVM is platform dependent because it is responsible to convert the bytecodes into the machine language for the specific computer/machine.

JRE

-It stands for Java Runtime environment. It contains the JVM, the library files and the other supporting files. To run a java program, the JRE must be installed in the system. So, we can simply say JRE=JVM+ some packages.

JDK

-It stands for Java Development Kit. It provides the tool which we need to develop java programs and JRE. This tools contains javac.exe, java.exe etc. When we launch a java application, it will open the JRE and load the class and then, in turn, it will execute the main method. So, we can conclude that JDK=JRE+ Development tools.

Bytecode

Bytecodes are machine language of the JVM. They provide the instruction set for a JVM. In other words, it is a virtual machine language in which java code is compiled. JVM comes into the picture because it stands between these bytecodes and our physical machine.

Platform

-We use the term platform to mean where the program will run. It can be your machine, your fully developed OS etc. When we say a language is platform independent, we mean that the code of a programmer will not vary across different platforms.

So once the java program is compiled, we get the bytecodes. These bytecode format is same for every platform (Windows/Linux/Solaris etc.).So, we need an interpreter who will interpret these bytecode and will produce the machine specific codes. Now JVM comes into the picture. Here in Java, these bytecodes are interpreted by JVM which is available for all OS. So, to port the java program into a new platform, we need to port the java interpreter.

So the pair -JVM and bytecode make Java portable.

Note: So the bottom line is that the trio- JVM, JRE and JDK are platform dependent (because of the OS dependence) but Java is platform independent.

We must remember the simple fact: Any machine language is dependent on the OS of the machine. So, if we have dependency on the machine specific OS, we are not platform independent. Java is platform independent because once the source code is compiled into standard bytecodes, those bytecodes are platform independent. Because of this facility Sun Microsystem is created the slogan WORA (Write Once Run Anywhere) for Java.

IDE

It stands for Integrated Development Environment. They provide the facilities for software development. In general, they are very smart- they provide us intelligent code completion technique. They can also highlight/suggest about different kinds of possible fixes in our code. An IDE should have a source editor, a debugger and the automation tools to build the application. IDE's, in general, contain a compiler or an interpreter (or both). *We have used eclipse here* which contains both of these.

Installation

We need two major things.

1. JDK

2. IDE

Visit the page:

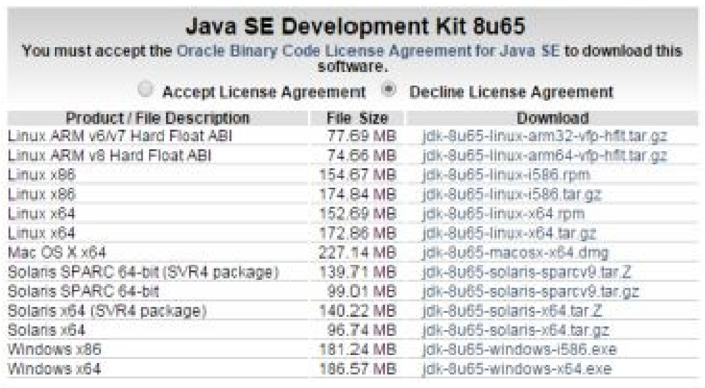
http://www.oracle.com/technetwork/java/javase/downloads/index.html

Or

To download JDK, directly go here:

http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

You’ll get a screen containing something like this:



Java SE Development Kit 8u65

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

[ ] Accept License Agreement [✓ Decline License Agreement]

Product / File Description	File Size	Download
Linux ARM v6/v7 Hard Float ABI	77.69 MB	jdk-8u65-linux-arm32-vfp-hflt.tar.gz
Linux ARM v8 Hard Float ABI	74.66 MB	jdk-8u65-linux-arm64-vfp-hflt.tar.gz
Linux x86	154.67 MB	jdk-8u65-linux-i586.rpm
Linux x86	174.84 MB	jdk-8u65-linux-i586.tar.gz
Linux x64	152.69 MB	jdk-8u65-linux-x64.rpm
Linux x64	172.86 MB	jdk-8u65-linux-x64.tar.gz
Mac OS X x64	227.14 MB	jdk-8u65-macosx-x64.dmg
Solaris SPARC 64-bit (SVR4 package)	139.71 MB	jdk-8u65-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	99.01 MB	jdk-8u65-solaris-sparcv9.tar.gz
Solaris x64 (SVR4 package)	140.22 MB	jdk-8u65-solaris-x64.tar.Z
Solaris x64	96.74 MB	jdk-8u65-solaris-x64.tar.gz
Windows x86	181.24 MB	jdk-8u65-windows-i586.exe
Windows x64	186.57 MB	jdk-8u65-windows-x64.exe

Try to download the latest version based on your system configuration (e.g. 32 bit/64 bit, Windows/Linux etc.)

To download eclipse IDE:

Go here.

https://eclipse.org/downloads/

As mentioned above try to download the latest version based on your system configuration.

Naming Conventions

Class- should start with an uppercase letter and should be a noun e.g. MyClass, String etc.

Interface/s- should start with an uppercase letter and should be an adjective e.g. Runnable, Remote

Method/s- They should start with a lowercase letter and be a verb e.g. main(), showMyMethod() etc.

Variable/s- They should start with lowercase letter e.g. myIntegerValue, myDoubleValue, myName etc.

Package/s- They should be all in lower case latter e.g. mypackage, package1 etc.

Constants- They should be in uppercase letters e.g. MY_CONSTANT etc.

Apart for few special cases, we have tried to maintain these conventions across the book.

Introduction

History:

In June 1991, James Gosling, Mike Sheridan and Patrick Naughton initiated the project of Java language. There was an Oak tree outside Gosling’s office. And people say that this is why, originally the language was named Oak. Later they renamed the project as Green (Their team name was also Green team). And finally they renamed it to Java. The Green project was chartered by Sun Microsystem.

The team members wanted such a name that will be very much unique in nature and at the same time, it should reflect the essence of upcoming technologies .So, they picked names like “Dynamic”, “Revolutionary”, “Silk”, “Jolt”, “DNA” etc.

James Gosling later told that Java was one of the top choices along with Silk. But finally they selected Java because most of the team mates liked this name.

Java became Open source on November13, 2006. Sun finished the process by making all of Java’s core code available under free software/open-source distribution terms, (aside from a small portion of code to which they did not hold the copyright) on May 08, 2007.

Later Oracle Corporation purchased Sun Microsystem and the acquisition process was finished on January 27, 2010.

Primary objective:

These qualities were the primary focus area for Java:

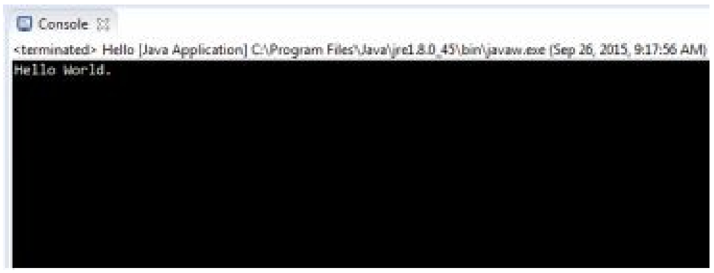
1. Simple, Object-oriented, and familiar.
2. Robustness and Security
3. Architecture-neutral and Portable.
4. High Performance capabilities.
5. Interpreted, Threaded, and Dynamic.

Our First program:

Now let us go through our first program. We’ll print *Hello World* here.

```
package javaclassnotes.programs;

public class HelloWorld
{
    public static void main(String args[])
    {
                System.out.println("Hello World.");
    }
}
```

Output:



Console

```
<terminated> Hello [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Sep 26, 2015, 9:17:56 AM)
Hello World.
```

Analysis:

1. *First of all, throughout the book, we have organized the programs into package/s. But for this program, it was not mandatory. Once we go through the chapters on package, it will be clear to us.*

2. It is the basic structure of the main method. The meaning and significance of each keyword will be clear to you gradually. So, for the time being, you must follow this structure.

3. Our source file name is HelloWorld.java. We need to use .java extension for our java files. It is the requirement for the compiler.

4. Java is case-sensitive.

5.

main- The program will start from here.

public- The access specifier. Access specifiers are used to control the visibility of the members.

static- It allows us to call main() without instantiate a particular instance of the class. We'll analysis "static" later.

void-return type.

String args[]-args is an array of instances of String class. String Objects are used to store character strings.

println-It is used to display information.

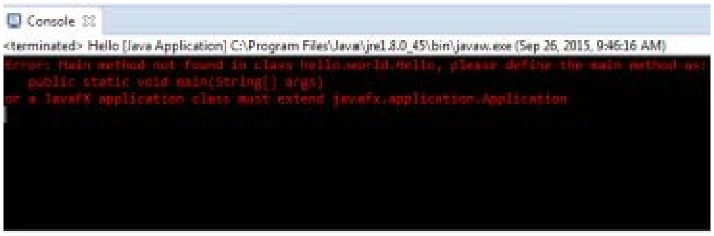
System.out- Difficult to explain at this point. Just we can know that System is a class and out is output stream associated with the console.

Quiz:

What will be the output?

```
class Hello {
    static void main(String args[])
    {
                System.out.println("Hello World.");
    }
}
```

Output:



Console

```
<terminated> Hello [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Sep 26, 2015, 9:46:16 AM)
Error: Main method not found in class hello.world.Hello, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

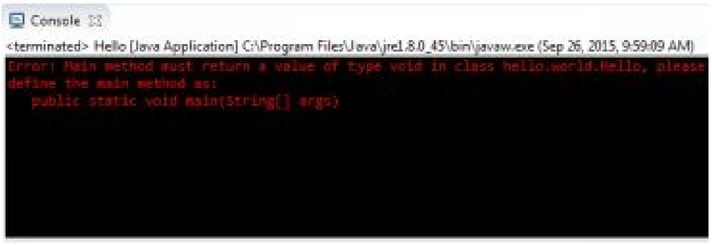
So, we cannot omit the keyword public here.

Quiz:

What will be the output?

```
class Hello
{
    public static int main(String args[])
    {
                System.out.println("Hello World.");
                return 0;
    }
}
```

Output:



Console

```
<terminated> Hello [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Sep 26, 2015, 9:59:09 AM)
Error: Main method must return a value of type void in class hello.world.Hello, please
define the main method as:
   public static void main(String[] args)
```

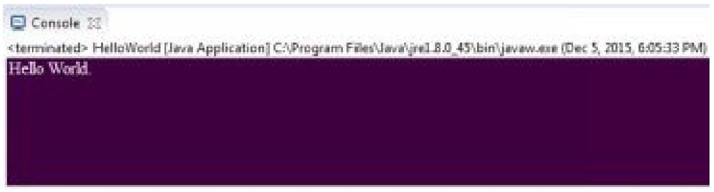
So, remember that the return type of main method should be void.

Quiz:

What will be the output?

```
class HelloWorld
{
    static public void main(String args[])
    {
                System.out.println(“Hello World.”);
    }
}
```

Output:



Console

```
<terminated> HelloWorld [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Dec 5, 2015, 6:05:33 PM)
Hello World.
```

So, we can see that we can change the order. Instead of public static void main(…) , we can write: static public void main(…).But we’ll always use the convention.

Class

Class:

A class is a blueprint or a template. It will describe about the behaviors of its objects.

Object:

An object is an instance of a class.

Object Oriented Programming (OOP) is based on these two concepts. With a class, we are creating a new datatype and objects are used to hold the data (fields) and methods. Object behavior can be exposed through these methods.

Suppose, we say, Sachin is a Cricketer. If we have some idea about cricket, we can predict that either Sachin plays as a batsman or as a bowler or as a wicketkeeper (or as an all-rounder).*Here Cricketer is a class and Sachin can be considered as an object of that class.*

Now come back to our Cricketer class again. Let us say, Sourav is a cricketer. Like the same manner, we can predict Sourav is a batsman or a bowler or a wicketkeeper. *Now we can see both Sachin and Sourav are objects of Cricketer Class but they have individual identity. Obviously Sourav and Sachin shows their skills in the game differently even though they are participating in the same game.*

Consider a different domain. *We can consider our pet dog or cat as an object of an Animal class.*

Now come to the programming. Suppose our class name is A. Then we can create an object "obA" of the class A with the following statement:

A obA=new A();

Actually, the above line can be decomposed of two lines as below:

A obA;

obA=new A();

Initially obA is a reference. Till this point, there is no memory allocated. But once the new comes into picture, the memory is allocated.

You must note that in the second line, class name is followed by a parentheses. These are for constructors. Constructors are used to describe what will happen when an object will be created. Constructors can have different attributes. But if our class does not specifically define a constructor, Java will supply a default one. In the above example, we have used a default constructor.

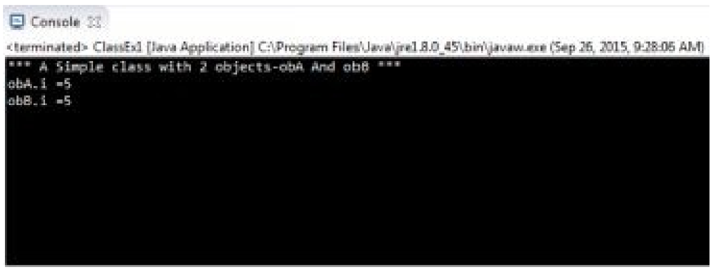
A simple class demonstration:

Here our class name is ClassA. It has only one field-i which is of type int. Here the value of i already has the value 5 associated with it. So, we can predict that if we create an object for this class, the object of that class will have an integer named i and the value of i in it will be 5.

For your ready reference, we have created 2 objects obA and obB for our class ClassA here. We have tested the values of i inside the objects. You can see that both have the value 5.

```
class ClassA
{
    int i=5;
}
class ClassEx1
{
    public static void main(String args[])
    {
            System.out.println(“*** A Simple class with 2 objects-obA And obB ***”);
            ClassA obA=new ClassA();
            ClassA obB=new ClassA();
            System.out.println(“obA.i =”+ obA.i);
            System.out.println(“obB.i =”+ obB.i);

    }
}
```

Output:



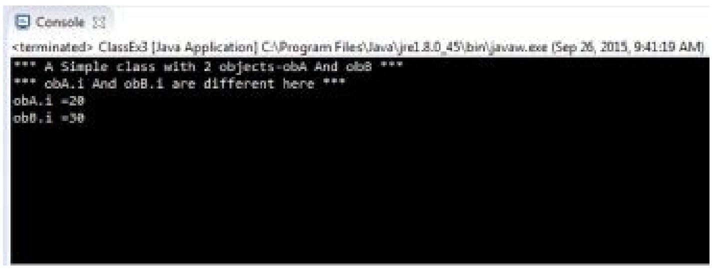
```
Console
<terminated> ClassEx1 [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Sep 26, 2015, 9:28:06 AM)
*** A Simple class with 2 objects-obA And obB ***
obA.i =5
obB.i =5
```

Class demonstration-2:

We have provide our own constructor here. We can see that now we can initialize objects with different values. obA has initialized integer i with the value 20 and obB has initialized the integer i with the value 30.

```
class ClassA
{
    int i;
    ClassA(int i)
    {
      this.i=i;
    }
}
class ClassEx3
{
    public static void main(String args[])
    {
              System.out.println("*** A Simple class with 2 objects-obA And obB ***");
              System.out.println("*** obA.i And obB.i are different here ***");
              ClassA obA=new ClassA(20);
              ClassA obB=new ClassA(30);
              System.out.println("obA.i ="+ obA.i);
              System.out.println("obB.i ="+ obB.i);

    }
}
```

Output:



Console

```
<terminated> ClassEx3 [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Sep 26, 2015, 9:41:19 AM)
*** A Simple class with 2 objects-obA And obB ***
*** obA.i And obB.i are different here ***
obA.i =20
obB.i =30
```

Students ask:

Sir, what is use of this here?

Good question. "this" is used to refer the current object. We can omit the use of this if we write the code like this:

```
class ClassA
{
    int i;//instance variable
    ClassA(int myInt)//myInt-local variable
    {
     i=myInt;
    }
}
```

We are familiar with the code like this: a=5; here we are assigning 5 into a. But can we write 5=a; ? No. Compiler will raise an issue.

Here myInt is our *local variable* (seen inside methods, blocks or constructors), i is our *instance variable* (declared inside a class but outside a method, block or constructor)

So, instead of myint, if we use i, we need to tell compiler about our intention. It should not be confused about *"which value is assigned where". Here we are assigning the value of the local variable to the instance variable and compiler should clearly understand our intention. With this.i=i; compiler will clearly understand the value of the local variable i is assigned to instance variable i.*

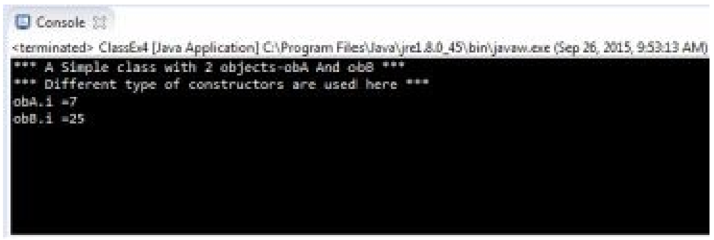
Class demonstration-3:

Here we have used two constructors. Go through the program. Notice that we can initialize an object with a default value here. If the default constructor is used during the creational process of an object, the instance variable i will be initialized with 7. We can also supply different values through the non-default constructor.

```
class ClassA
{
    int i;
    ClassA()
    {
            this.i=7;
    }
    ClassA(int i)
    {
            this.i=i;
    }
}
class ClassEx3
```

```
{
    public static void main(String args[])
    {
            System.out.println("*** A Simple class with 2 objects-obA And obB ***");
            System.out.println("*** Different type of constructors are used here ***");
            ClassA obA=new ClassA();
            ClassA obB=new ClassA(25);
            System.out.println("obA.i ="+ obA.i);
            System.out.println("obB.i ="+ obB.i);
    }
}
```

Output:



```
Console
<terminated> ClassEx4 [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Sep 26, 2015, 9:53:13 AM)
*** A Simple class with 2 objects-obA And obB ***
*** Different type of constructors are used here ***
obA.i =7
obB.i =25
```

Assignment:

1. Create a class Vehicle. The class should have two fields-no_of_seats and no_of_wheels. Create two objects-Motorcycle and Car for this class. Your output should show the descriptions for Car and Motorcycle.

Inheritance

The main objective of inheritance is to promote reusability and eliminate redundancy (of code).Here a child class obtains the features of its parent class. By parent class we mean the class which is at the higher level in the class hierarchy compared to another class (which is termed as a child class).

Types:

In general, we deal with 4 types of inheritance.

Single inheritance: One child class is derived from one base class.

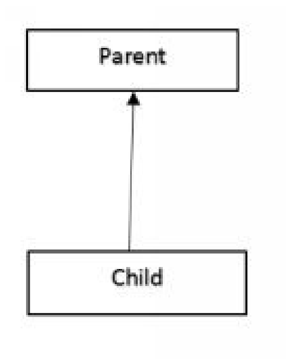


Figure: Single inheritance

The format of code is like this:

```
 class Parent
{
//your code…
}
class Child extends Parent
{
//your code…
}
```

Hierarchical inheritance: Multiple child class can be derived from one base class.

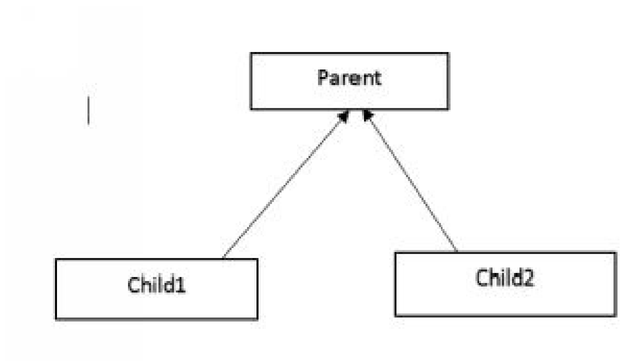


Figure: Hierarchical inheritance

The format of code is like this:

```
 class Parent
  {
        //your code…
  }
  class Child1 extends Parent
  {
        //your code…
  }
  class Child2 extends Parent
  {
        //your code…
  }
```

Multilevel inheritance: Here the parent class has the grandchild.

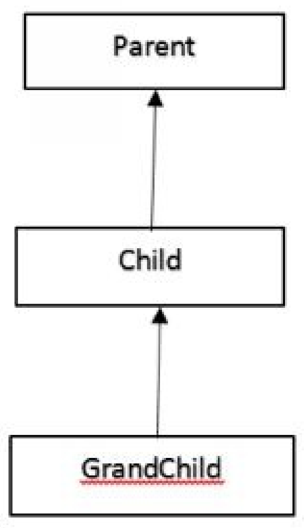


Figure: Multilevel inheritance

Teacher asks:

Now try to implement the concept with Java code.

Solution:

The format of code is like this:

```
  class Parent
```

```
   {
         //your code…
   }
   class Child extends Parent
   {
         //your code…
   }
   class GrandChild extends Child
   {
         //your code…
   }
```

Multiple inheritance: Here a child can derive from multiple Parents.

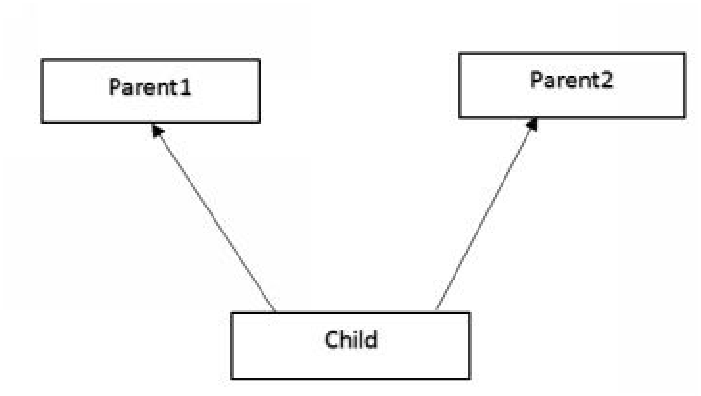


Figure: Multiple inheritance

Note that, Java does not support this type of inheritance (through class).i.e.in Java, a child class cannot derive from more than one parent class. To deal with this type of situation we need to understand interfaces.

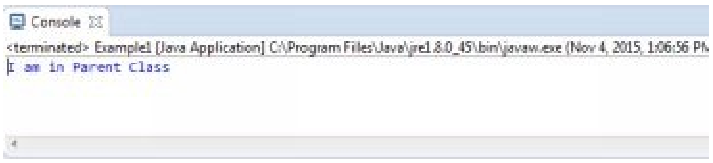
Note: There is another type of inheritance which is termed as *hybrid inheritance*. This is a combination of one or more types of the above inheritance/s.

A simple program on Inheritance:

```
class ParentClass
{
public void show()
  {
    System.out.println("I am in Parent Class");
  }
}
```

```
class ChildClass extends ParentClass
{
}

class Example1
{
    public static void main(String args[])
    {
                ChildClass child1=new ChildClass();
    //Calling  show() through ChildClass object
                child1.show();
    }
}
```

Output:



```
<terminated> Example1 [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 4, 2015, 1:06:56 PN
I am in Parent Class
```

Students ask:

Why Java does not support multiple inheritance through class?

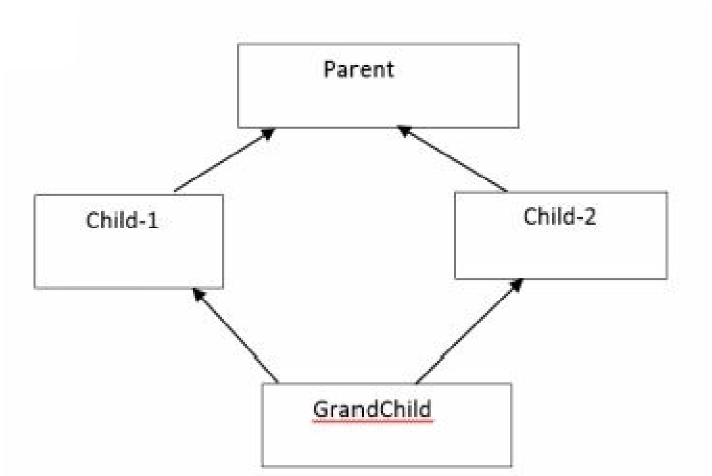
The main reason is to avoid ambiguity. They can cause some confusion in some typical scenarios like this:

Suppose in our parent class we have a method named show(). The parent class has multiple children-say child1 and child 2 who are overriding the method differently for their own purpose. The code may look like this:

```
    class Parent
    {
    public void show()
    {
      System.out.println  ("I am in Parent");
    }
    }
    class Child1 extends  Parent
```

```
{
public void show()
{
  System.out.println  ("I am in Child-1");
}
}
class Child2 extends  Parent
{
public void show()
{
  System.out.println  ("I am in Child-2");
}
}
```

Now say our Grandchild derives from both Child1 and Child2 but it has not overridden the method show().



So, now we have an ambiguity-From which class, GrandChild will inherit/call show()- Child1 or Child 2.In order to remove this type of ambiguity Java does not support multiple inheritance through class. *This problem is known with a famous name- the Diamond problem.*

Teacher asks:

Can we have Hybrid inheritance in Java?

Interesting question. Think carefully. Hybrid inheritance can be a combination of two or more type of the above inheritance/s. *So, the answer to this question is yes till the point where we are not trying to combine any multiple inheritance through class.* And if our intention is to make such a hybrid inheritance in which we need to have any kind of multiple inheritance (through class), Java will not support that concept.

Note:

Remember that in Java, Object (in java.lang package) is the superclass for all classes. Because all other classes directly or indirectly is an inheritor of that class.

Assignments:

1. Write a Simple Program to implement Hierarchical Inheritance.
2. Write a Simple Program to implement Multilevel Inheritance.

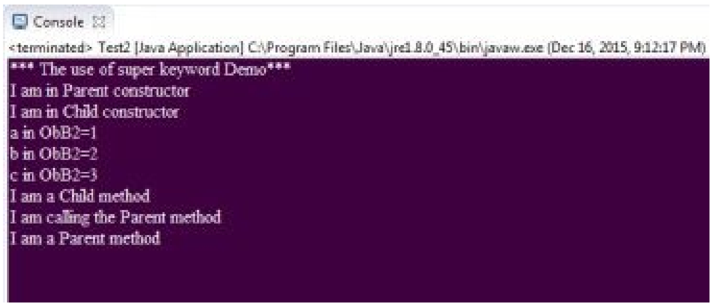
A special keyword: super

In Java, we have a special keyword-super. It is used to access the members of the parent class (super class) in an efficient way. Whenever a child class wants to refer its immediate parent, it should use this keyword.

We can examine the use of super with this simple example.

```
package javaclassnotes.testprograms;

class A2
{
   int a;
   int b;
   A2(int a,int b)
   {
            System.out.println("I am in Parent constructor");
            this.a=a;
            this.b=b;
   }
   void parentMethod()
   {
            System.out.println("I am a Parent method");
   }
}
class B2 extends A2
{
    int c;
    B2(int a, int b,int c)
    {
            super(a,b);
            System.out.println("I am in Child constructor");
            this.c=c;
    }
    void childMethod()
    {
            System.out.println("I am a Child method");
            System.out.println("Now I am going to call the Parent method");
            super.parentMethod();
    }

}

class Test2
{
    public static void main(String args[])
    {
            System.out.println("*** The use of super keyword Demo***");
            B2 obB2=new B2(1,2,3);
            System.out.println("a in ObB2="+ obB2.a);
            System.out.println("b in ObB2="+ obB2.b);
            System.out.println("c in ObB2="+ obB2.c);
            obB2.childMethod();
    }
}
```

Output:



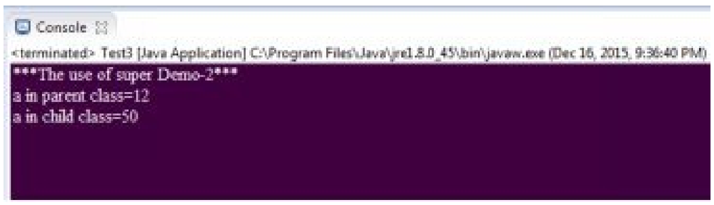
```
<terminated> Test2 [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Dec 16, 2015, 9:12:17 PM)
*** The use of super keyword Demo***
I am in Parent constructor
I am in Child constructor
a in ObB2=1
b in ObB2=2
c in ObB2=3
I am a Child method
I am calling the Parent method
I am a Parent method
```

We’ll examine another use of super with the following example. *Here we’ll see that even if the instance variable of the parent class becomes hidden by the child class’s instance variable, super can allow us to access the instance variable in super class.*

```
package javaclassnotes.testprograms;


class A3
{
    int a;
    A3()
    {
            a=25;//some default value
    }
}
class B3 extends A3
{
    int a;//this will hide a in A3
    B3()
    {
            super.a=12;//for a in parent class
            a=50;//for a in B(child class)
    }
    void display()
    {
            System.out.println(“a in parent class=”+ super.a);
            System.out.println(“a in child class=”+ a);
    }
}
```

```
class Test3
{
    public static void main(String args[])
    {
            System.out.println("***The use of super Demo-2***");
            B3 obB3=new B3();
            obB3.display();
    }
}
```

Output:



```
<terminated> Test3 [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Dec 16, 2015, 9:36:40 PM)
***The use of super Demo-2***
a in parent class=12
a in child class=50
```

Note:

Students ask:

***Can we use super keyword* to call methods that are hidden by a subclass?**

Yes.

Overloading

Teacher asks:

Consider the below program segments. Do you notice any specific pattern?

```
            int sum(int x,int y)
    {
             return x+y;
    }
    double sum(double x,double y)
    {
             return x+y;
    }
    String sum(String s1,String s2)
    {
             return s1.concat(s2);
    }
```

Students respond:

Yes sir. We are seeing all of the methods have the same name "sum" but from their method bodies it appears that each method is doing different things.

Teacher says: yes. You are correct. When we do this kind of coding, we term it as method overloading. *But you should notice that though method names are same but method signatures are different here.*

Students ask:

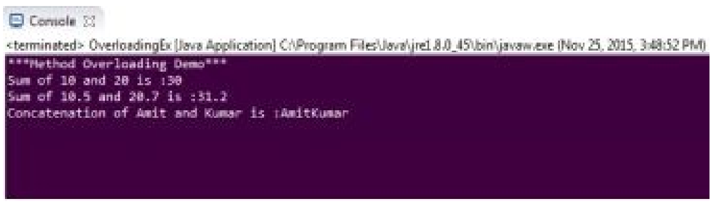
What is method signature?

Ideally method name with number and types of the parameters consist the method signature. Java compiler can distinguish among methods with same name but different parameter list. So, for Java compiler, double sum(**double** x, **double** y) is different from int sum(**int** x, **int** y).

Consider the below program. Here we represent method overloading with the following example:

```
package javaclassnotes.programs;
class Addition
{
    int sum(int x,int y)
    {
            return x+y;
    }
    double sum(double x,double y)
    {
            return x+y;
    }
    String sum(String s1,String s2)
    {
            return s1.concat(s2);
    }
}

public class OverloadingEx
{
    public static void main(String args[])
    {
            System.out.println("***Method Overloading Demo***");
            Addition additionOb=new Addition();
            int sumOfIntergers=additionOb.sum(10,20);
            System.out.println("Sum of 10 and 20 is :"+sumOfIntergers);
            double sumOfDoubles=additionOb.sum(10.5,20.7);
            System.out.println("Sum of 10.5 and 20.7 is :"+sumOfDoubles);
            String sumOfStrings=additionOb.sum("Amit","Kumar");
            System.out.println("Concatenation of Amit and Kumar is :"+sumOfStrings);
    }
}
```

Output:



```
<terminated> OverloadingEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 25, 2015, 3:48:52 PM)
***Method Overloading Demo***
Sum of 10 and 20 is :30
Sum of 10.5 and 20.7 is :31.2
Concatenation of Amit and Kumar is :AmitKumar
```

Teacher asks:

Is it an example of method overloading?

int sum(**int** x,**int** y)

```
    {
            return x+y;
    }
int sum(int x,int y,int z)
    {
            return x+y+z;
    }
```

Answer: Yes.

Teacher asks:

Is it an example of method overloading?

```
int sum(int x,int y)
    {
            return x+y;
    }
double sum(int x,int y)
    {
            return x+y;
    }
```

Answer: No. Compiler will not consider "return type" to differentiate these methods. *Return type is not considered as a part of method signature.*

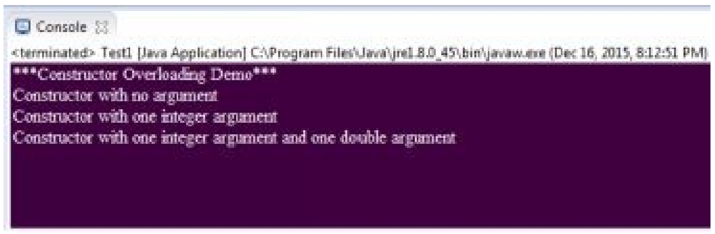
Students ask:

Sir, can we have constructor overloading?

Definitely. You can write a similar program for constructor overloading.

```
package javaclassnotes.testprograms;

class A1
{
    A1()
    {
            System.out.println("Constructor with no argument");
    }
    A1(int a)
    {
            System.out.println(“Constructor with one integer argument”);
    }
    A1(int a,double b)
    {
            System.out.println(“Constructor with one integer argument and one double
argument”);
    }
}
class Test1
{
    public static void main(String args[])
    {
            System.out.println(“***Constructor Overloading Demo***”);
            A1 ob1=new A1();
            A1 ob2=new A1(2);
            A1 ob3=new A1(2,3.7);
    }
}
```

Output:



```
Console
<terminated> Test1 [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Dec 16, 2015, 8:12:51 PM)
***Constructor Overloading Demo***
Constructor with no argument
Constructor with one integer argument
Constructor with one integer argument and one double argument
```

Students ask:

Sir, it appears to me that it is also method overloading. What is the difference between a constructor and a method?

Notice carefully. A constructor has the same name as class and also it has no return type. *So, you can consider a constructor as a special kind of method which has the same name as class and no return type. But there are many other difference: the main focus of a constructor is to initialize objects. They cannot be called directly.*

Students ask:

So Sir, can we write code like this?

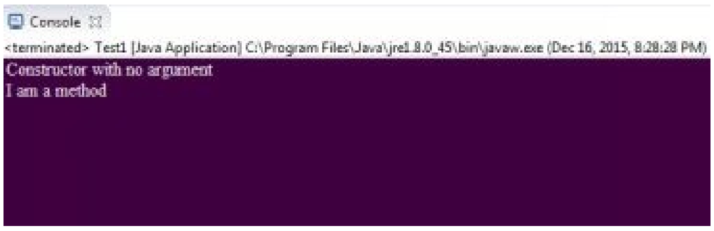
```
class A1
```

```
{
    //It is a constructor. It has no return type.
    A1()
    {
            System.out.println("Constructor with no argument");
    }
    //It is a method. It has return types.
    void A1()
    {
            System.out.println("I am a method");
    }
}
```

Sure. Now, the following lines inside main function

```
    A1 ob1=new A1();
            ob1.A1(5);
```

can create output like this:



Console ☒

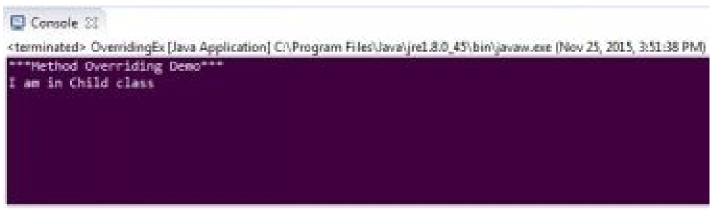
```
<terminated> Test1 [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Dec 16, 2015, 8:28:28 PM)
Constructor with no argument
I am a method
```

Overriding

Sometimes we want to redefine or modify the behavior of our parent class. Method overriding comes into picture in such a scenario. Consider the below program. Note that, here showMe() method has the same signature in both the parent class and its child class.

```
package javaclassnotes.programs;
class ParentClass
{
   public void showMe()
   {
            System.out.println("I am in Parent class");
   }
}
class ChildClass extends ParentClass
{
   public void showMe()
   {
            System.out.println("I am in Child class");
   }
}
class OverridingEx
{
   public static void main(String args[])
   {
            System.out.println("***Method Overriding Demo***");
            ChildClass childOb=new ChildClass();
            childOb.showMe();
   }
}
```

Output:



```
<terminated> OverridingEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 25, 2015, 3:51:38 PM)
***Method Overriding Demo***
I am in Child class
```

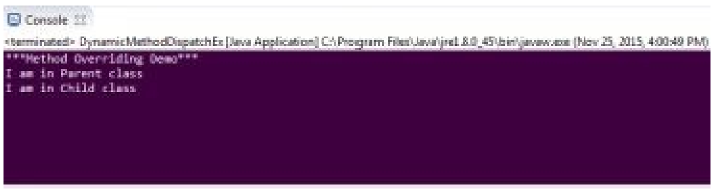
Dynamic Method Dispatch:

This is an extremely important concept in Java. *Java can implement runtime polymorphism through this technique. This technique is considered to implement runtime polymorphism because the call to an overridden method is resolved dynamically at runtime*. Java will call the appropriate method based on the object which we are referring.

```
package javaclassnotes.programs;
class MyParentClass
{
   public void showMe()
   {
         System.out.println("I am in Parent class");
   }
}
class MyChildClass extends MyParentClass
{
   public void showMe()
   {
         System.out.println("I am in Child class");
   }
}
class DynamicMethodDispatchEx
{
   public static void main(String args[])
   {
         System.out.println("***Dynamic Method Dispatch Demo***");
         MyParentClass parent=new MyParentClass();
         parent.showMe();
         MyChildClass childOb=new MyChildClass();
         /*Parent class reference to a child object*/
         parent=childOb;
```

```
                childOb.showMe();
    }
}
```

Output:



Console

```
<terminated> DynamicMethodDispatchEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 25, 2015, 4:00:49 PM)
***Method Overriding Demo***
I am in Parent class
I am in Child class
```

Points to remember:

Through a parent class reference, we can refer a child class object but the reverse is not applicable.

So,

MyParentClass parent=**new** MyChildClass(); is ok but

MyChildClass child=**new** MyParentClass(); will raise error.

Teacher asks:

Now there may be some situation where we want a restriction: A method in the parent should not be overridden by its child. How can we achieve that?

In many interviews, you can face this question. We must remember that we can prevent overriding by the use of static, private or final keywords. But here we discuss only the use of "final". It is very much helpful because compiler itself will prevent the process of overriding.

```
class ParentClass
{
    //Use of final to prevent overriding
    final public void showMe()
    {
                System.out.println("I am in Parent class");
    }
}
class ChildClass extends ParentClass
{
    //Cannot override now: It is not allowed
    public void showMe()
    {
                System.out.println(“I am in Child class”);
    }
}
```

So, with the above code, compiler will raise the error.

Abstract Class

These are incomplete classes and we cannot instantiate objects from this type of classes.

In general, if a class contains at least one incomplete/abstract method, the class itself is an abstract class. By the term “abstract method”- we mean that the method has the declaration (or signature) but no implementation.

The technique is useful when the super class can define a generalized form (that will be shared by its subclasses) and passes the responsibilities to fill the details to its subclasses.

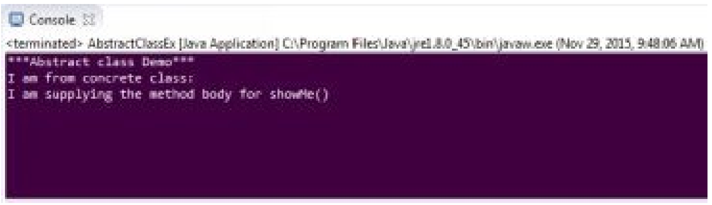
A simple abstract class demo:

Implementation-1:

```
package javaclassnotes.programs;

abstract class MyAbstractClass

{

    public  abstract void showMe();

}

class MyConcreteClass extends MyAbstractClass

{

    @Override

    public void showMe()

    {

             System.out.println(“I am from concrete class:”);

             System.out.println(“I am supplying the method body for showMe()”);


    }

}

class AbstractClassEx

{

    public static void main(String Args[])

    {

             System.out.println(“***Abstract class Demo***”);

             //Illegal:Cannot instantiate
```

```
            //MyAbstractClass abstractOb=new MyAbstractClass();
            MyConcreteClass concreteOb=new MyConcreteClass();
            concreteOb.showMe();
    }


}
```

Output:



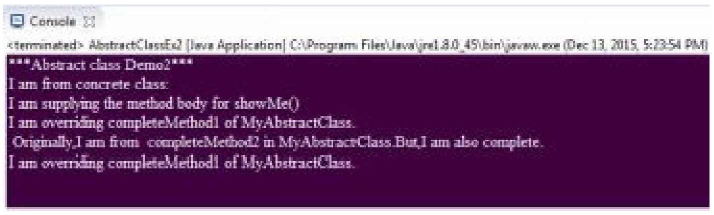
```
Console
<terminated> AbstractClassEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 29, 2015, 9:48:06 AM)
***Abstract class Demo***
I am from concrete class:
I am supplying the method body for showMe()
```

An abstract class can contain concrete methods also. The child class may or may not override those methods.

Implementation-2:

```
package javaclassnotes.programs;
abstract class AbstractClass
{
    public  abstract void showMe();
    public void completeMethod1()
    {
            System.out.println(" Originally,I am from completeMethod1 in
MyAbstractClass.But,I am complete.");
    }
    public void completeMethod2()
    {
            System.out.println(" Originally,I am from  completeMethod2 in
MyAbstractClass.But,I am also complete.");
    }
}
class ConcreteClass extends AbstractClass
{
    @Override
    public void showMe()
    {
            System.out.println("I am from concrete class:");
            System.out.println("I am supplying the method body for showMe()");


    }
    //It wants to override completeMethod1() in MyAbstractClass
    public void completeMethod1()
    {
            System.out.println("I am overriding completeMethod1 of
MyAbstractClass.");
    }
}
class AbstractClassEx2
{
    public static void main(String Args[])
    {
            System.out.println("***Abstract class Demo2***");
            ConcreteClass concreteOb=new ConcreteClass();
            concreteOb.showMe();
            //It will show that completeMethod1 is redefined in MyConcreteClass.
            concreteOb.completeMethod1();
            //It will show the details of completeMethod2 defined in MyAbstractClass.
            concreteOb.completeMethod2();
            //Following declaration will be fine
            AbstractClass abstractRef=new ConcreteClass();
            abstractRef.completeMethod1();
    }
}
```

Output:



```
<terminated> AbstractClassEx2 [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Dec 13, 2015, 5:23:54 PM)
***Abstract class Demo2***
I am from concrete class:
I am supplying the method body for showMe()
I am overriding completeMethod1 of MyAbstractClass.
 Originally,I am from  completeMethod2 in MyAbstractClass.But,I am also complete.
I am overriding completeMethod1 of MyAbstractClass.
```

Can we implement the concept of dynamic method dispatch here?

Yes. Following declaration will be perfectly fine and it will call CompleteMethod1 of the ConcreteClass.

```
AbstractClass abstractRef=new ConcreteClass();
abstractRef.completeMethod1();
```

Students ask:

Can an abstract class contain fields?

Yes.

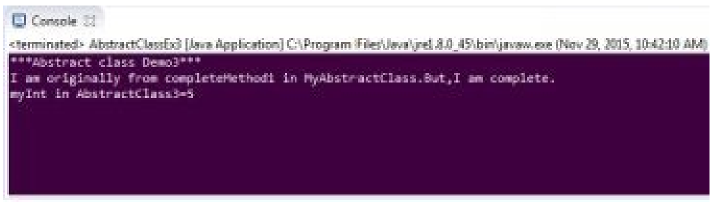
Following example will demonstrate how we can use the concept of dynamic method dispatch here. Also, the program will show that an abstract class contain fields.

Implementation-3:

```
package javaclassnotes.programs;
abstract class AbstractClass3
{
    public int myInt=5;
    public  abstract void showMe();
    public void completeMethod1()
    {
                System.out.println("I am originally from completeMethod1 in
MyAbstractClass.But,I am complete.");
    }
}
class ConcreteClass3 extends AbstractClass3
{
    @Override
    public void showMe()
    {
                System.out.println("I am from concrete class:");
                System.out.println("I am supplying the method body for showMe()");

    }
}
class AbstractClassEx3
```

```
{
    public static void main(String Args[])
    {
            System.out.println("***Abstract class Demo3***");
            AbstractClass3 abstractRef=new ConcreteClass3();
            abstractRef.completeMethod1();
            System.out.println("myInt in AbstractClass3="+abstractRef.myInt);

    }

}
```

Output:



```
Console
<terminated> AbstractClassEx3 [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 29, 2015, 10:42:10 AM)
***Abstract class Demo3***
I am originally from completeMethod1 in MyAbstractClass.But,I am complete.
myInt in AbstractClass3=5
```

Students ask:

Suppose in a class we have 50+ methods and out of that only one is an abstract method. Still we need to mark the class as abstract?

Yes. If a class contains at least one abstract method, the class itself is abstract. You can think from a general point of view-an abstract keyword is used in a sense to represent the incompleteness. So, if your class contains one incomplete method, your class itself is incomplete and hence need to mark by the keyword abstract.

So, the simple formula is: whenever your class has at least an abstract method, your class itself is an abstract class.

Teacher asks:

Now consider a reverse scenario. Suppose, you have marked your class abstract but there is no abstract method in it like this:

```
abstract class AbstractClass
{
    public void completeMethod1()
    {
            System.out.println("A complete method");
    }
    public void completeMethod2()
    {
            System.out.println("Another complete method.");
    }
}
```

Can we compile the program?

Yes. Still it will compile but till this point, you cannot create object for this class.

Students ask:

So sir, how you can create object from an abstract class?

We mentioned already that we cannot create objects from an abstract class.

Students ask:

Sir, it appears to me that an abstract class has virtually no use if it is not extended. Is the understanding correct?

Yes.

Students ask:

If a class extends an abstract class, it has to implement all the abstract methods?

It may or may not implement all the abstract methods in the parent class. The simple formula is that if you want to create objects of a class, the class needs to be complete i.e. it should not contain any abstract methods. *So, if the child class cannot provide implementation (i.e. body) of all the abstract methods, it should be marked again with the keyword abstract like the below example.*

```
abstract class AbstractClass
{
    public abstract void inCompleteMethod1();
    public abstract void inCompleteMethod2();
}
```

```
abstract class child1 extends AbstractClass
{
    @Override
    public void inCompleteMethod1()
    {
              System.out.println("Implementing the inCompleteMethod1()");


    }
}
```

Students ask:

A concrete class is a class which is not abstract-is the understanding correct?

Yes.

Students ask:

Can we tag a method with both abstract and final?

No. Just think, by declaring abstract, you want overriding and by declaring final, you want to prevent overriding.

Students ask:

Can we have constructor overriding in Java?

No.

Interface

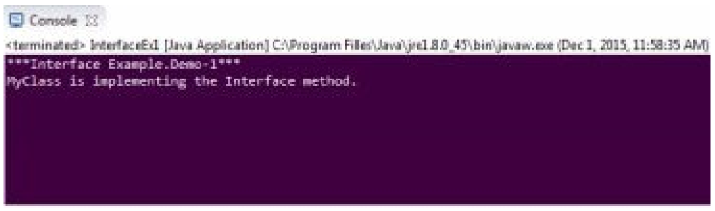
With the interface, we declare what we are going to implement but we are not specifying how we are going to that. These are similar to classes but with no instance variables and all of their methods are declared without a body (i.e. methods are actually abstract).

We can support dynamic method resolution during run time with the help of interfaces. Once defined, a class can implement any number of interfaces.

Implementation-1:

```
package javaclassnotes.programs;
interface MyInterface
{
    void show();
}
class MyClass implements MyInterface
{
    @Override
    public void show()
    {
            System.out.println(“MyClass is implementing the Interface method.”);

    }
}

public class InterfaceEx1
{
    public static void main(String args[])
    {
            System.out.println(“***Interface Example.Demo-1***”);
            MyClass myClassOb=new MyClass();
            myClassOb.show();
    }

}
```

Output:



```
<terminated> InterfaceEx1 [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Dec 1, 2015, 11:58:35 AM)
***Interface Example.Demo-1***
MyClass is implementing the Interface method.
```

Implementation-2:

Interface has two methods. But a class is implementing only one. Then the class itself becomes abstract.

```
package javaclassnotes.programs;


interface MyInterface2
{
    void show1();
    void show2();
}
abstract class MyClass2 implements MyInterface2
{
    @Override
    public void show1()
    {
                System.out.println("MyClass2 is implementing the show1()
method.");
    }
}
```

Note: So the formula is: A class needs to implement all the methods defined in the interface. Otherwise, it will be an abstract class.

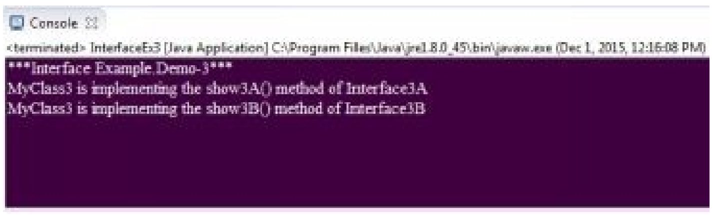
Implementation-3:

A class is implementing multiple interfaces.

```
package javaclassnotes.programs;
interface MyInterface3A
{
    void show3A();
}
```

```
interface MyInterface3B
{
    void show3B();
}
class MyClass3 implements MyInterface3A,MyInterface3B
{
    @Override
    public void show3A()
    {
                System.out.println("MyClass3 is implementing the show3A() method of
Interface3A");
    }
    @Override
    public void show3B() {
                System.out.println("MyClass3 is implementing the show3B() method of
Interface3B");


    }
}
public class InterfaceEx3 {
    public static void main(String args[])
    {
                System.out.println("***Interface Example.Demo-3***");
                MyClass3 myClassOb=new MyClass3();
                myClassOb.show3A();
                myClassOb.show3B();
    }
}
```

Output:



```
<terminated> InterfaceEx3 [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Dec 1, 2015, 12:16:08 PM)
***Interface Example Demo-3***
MyClass3 is implementing the show3A() method of Interface3A
MyClass3 is implementing the show3B() method of Interface3B
```

Students Ask:

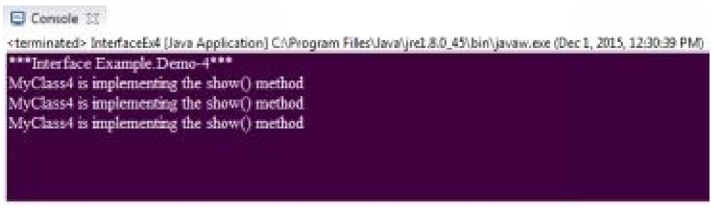
In the above program, method names were different in interfaces. But if both of the interfaces contain the same method name, can we implement them?

Go through the following implementation.

Implementation-4:

```java
package javaclassnotes.programs;

//Both of the interface have the same method name “show()”.
interface MyInterface4A
{
    void show();
}
interface MyInterface4B
{
    void show();
}
class MyClass4 implements MyInterface4A,MyInterface4B
{
    @Override
    public void show()
    {
            System.out.println(“MyClass4 is implementing the show() method “);

    }
}
public class InterfaceEx4 {
    public static void main(String args[])
    {
            System.out.println("***Interface Example.Demo-4***");

            //All the 3 callings are legal.
            MyClass4 myClassOb=new MyClass4();
            myClassOb.show();

            MyInterface4A inter4A=myClassOb;
            inter4A.show();

            MyInterface4B inter4B=myClassOb;
            inter4B.show();

    }
}
```

Output:



Console

```
<terminated> InterfaceEx4 [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Dec 1, 2015, 12:30:39 PM)
***Interface Example.Demo-4***
MyClass4 is implementing the show() method
MyClass4 is implementing the show() method
MyClass4 is implementing the show() method
```

Students ask:

Can an interface extend/implement another interface?

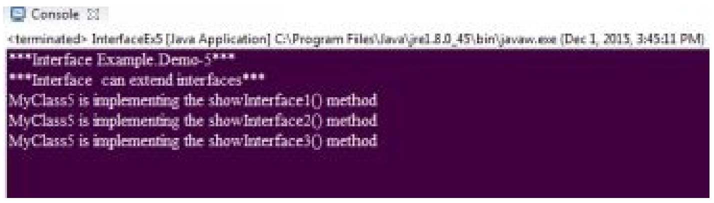
It can extend but not implement (by definition).

Implementation-5:

```
package javaclassnotes.programs;

interface Interface1
```

```
{
    void showInterface1Method();
}
interface Interface2
{
    void showInterface2Method();
}
//Interface extending another interfaces
interface Interface3 extends Interface1,Interface2
{
    void showInterface3Method();
}
class MyClass5 implements Interface3
{
    //Now MyClass5 needs to implement methods from Interface1,Interface2 and
Interface3
    @Override
    public void showInterface1Method() {
                System.out.println("MyClass5 is implementing the showInterface1() method
");


    }
    @Override
    public void showInterface2Method() {
                System.out.println("MyClass5 is implementing the showInterface2() method
");
    }
    @Override
    public void showInterface3Method() {
                System.out.println("MyClass5 is implementing the showInterface3() method
");


    }
}
public class InterfaceEx5 {
    public static void main(String args[])
    {
              System.out.println("***Interface Example.Demo-5***");
              System.out.println("***Interface  can extend interfaces***");
              MyClass5 myClassOb=new MyClass5();
              myClassOb.showInterface1Method();
              myClassOb.showInterface2Method();
              myClassOb.showInterface3Method();
    }
}
```

Output:



```
Console ☒
<terminated> InterfaceEx5 [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Dec 1, 2015, 3:45:11 PM)
***Interface Example.Demo-5***
***Interface  can extend interfaces***
MyClass5 is implementing the showInterface1() method
MyClass5 is implementing the showInterface2() method
MyClass5 is implementing the showInterface3() method
```

Tag/Tagging interface:

An interface which is empty is termed as a tag/tagging interface.

```
//tagging interface
interface ITaggingInterface
{
}
```

Teacher asks:

Can you tell me: Why we need a tagging interface?

1. We can create a common parent.

2. A class can claim membership in the set e.g. if our class implements the Serializable interface, it becomes serializable. So, our class actually becomes an interface type through polymorphism. Even a class that is implementing a tagging interface, need not define any

new method because the interface itself does not have any such method.

Teacher asks:

Can you tell me: What is the difference between an abstract class and an Interface?

1. An abstract class can have concrete methods in it but an interface cannot have that. [We'll come to this point later. Now in Java 8, we have a keyword called "default". We can use this keyword in an interface to provide some default implementation, see below in our implementation-6].

2. An abstract class can have only one parent class (can extend from another abstract class or concrete class), an interface can have multiple parent interfaces. An interface can extend from other interface/s only.

3. Members of an interface is by default public. An abstract class can have other flavors e.g. private, protected etc.

4. Variables in an interface is by default static final. An abstract class can have non-final variables.

Students ask:

Sir, then how we decide-whether we should use an abstract class or an interface?

Good question. I believe that if we want to have some centralized or default behavior/s, abstract class is a better choice .Because here we can provide some default implementation(in case of abstract class).On the other hand, interface implementation starts from a scratch. They indicate some kind of rules-what to be done (e.g. you must implement the method) but they will not enforce you how to be done. Also interfaces are preferred when we are trying to implement the concept of multiple inheritance.

But at the same time we also remember that if we need to add a new method in an interface, then we need to track down all the implementation/s of that interface and we need to put the concrete implementation for that method in all those places. An abstract class is ahead here-we can add a new method in an abstract class with a default implementation and our existing code can run smoothly.

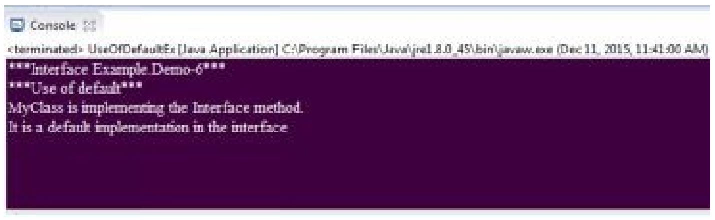
So, now Java has taken special care to this point and Java 8 has introduced the use of default keyword. In Java 8, we can prefix the word default before our intended method signature and can provide a default implementation. Interface methods are public by default, so, we do not need to mark it by the keyword public.

Implementation-6:

Consider the below implementation:

```
package javaclassnotes.programs;
interface MyDefaultInterface
{
   void show();
   default void defaultMethod()
   {
            System.out.println(“It is a default implementation in the interface”);
   }
}
class MyClass6 implements  MyDefaultInterface
{
   @Override
   public void show()
   {
            System.out.println(“MyClass is implementing the Interface method.”);

   }
}

public class UseOfDefaultEx
{
   public static void main(String args[])
   {
            System.out.println(“***Interface Example.Demo-6***”);
            System.out.println(“***Use of default***”);
            MyDefaultInterface interfaceOb=new MyClass6();
            interfaceOb.show();
            interfaceOb.defaultMethod();
   }
}
```

Output:



```
<terminated> UseOfDefaultEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Dec 11, 2015, 11:41:00 AM)
***Interface Example.Demo-6***
***Use of default***
MyClass is implementing the Interface method.
It is a default implementation in the interface
```

Assignment:

You have two classes- A and B. Class A is containing an abstract method showA().You also have an Interface called Inter. In Inter, you have a method showInter().Now write a simple program where B will implement the methods defined in A and Inter.

Package

Consider a simple scenario. Can you use the same class name twice in a java file? No. Compiler will raise the issue and it will point towards this naming collision. So we need to choose unique naming conventions each time whenever we are going to define a class. But we must remember that in real world programming, class name should be meaningful enough and so there is a possibility that two different programmer in a project are going to choose the same name for their class. Then how we can deal with those situations? Package will rescue us in those scenarios.

We can bundle our classes/interfaces etc. inside our own packages. Packages help us to avoid naming conflicts and/or to control the visibility. We can control the visibility inside a package in such a way that our particular class may or may not be exposed to outside world (both inside and outside packages).

Packages are reflected as directories. Creating a package in eclipse is quite easy. We do not even think about how Java runtime is going to find the proper packages or classes inside it. Otherwise, we need to put special attention to the CLASSPATH environment variable.

We must remember about the following points:

1. Package statement should be on top of our source file. If we do not explicitly define this statement, then all the classes/interfaces etc. will be in the current default package.

2. When one class refers another class inside the same package, package statement need not to be included.

Otherwise we need to use fully qualified class name like <packagename>.<classname> or we need to use import statement.

3. Whole package can be imported like:

import <packagename>.;*

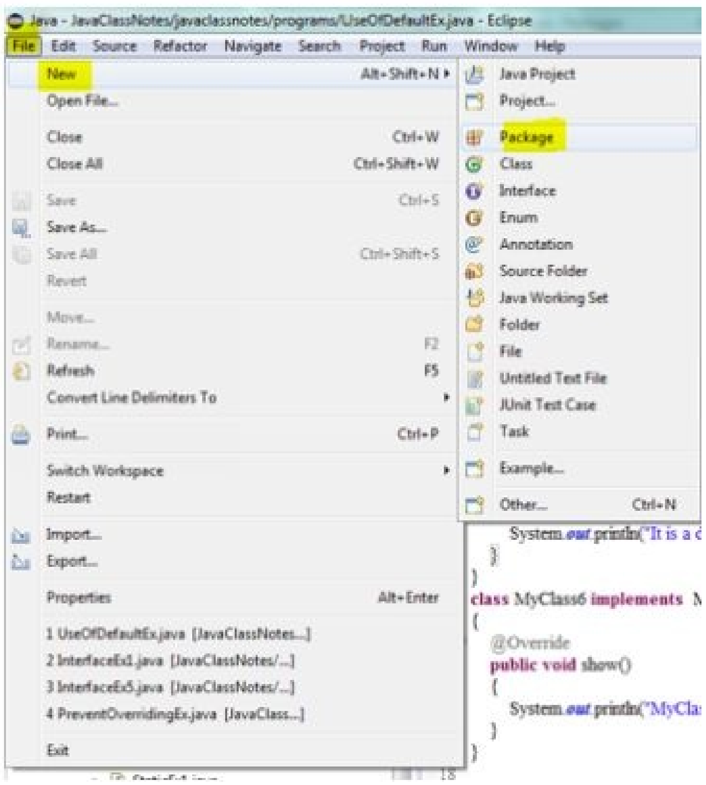
Or if we want to import only a particular class from a package, use:

import <packagename>.<classname>

4. The name of the package must follow the directory structure.

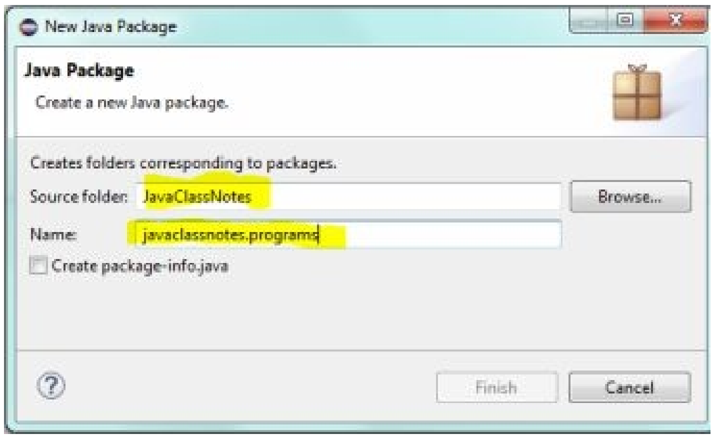
Creating a Package in Eclipse IDE:

1. Click File menu -> New ->Package



```
        System.out.println("It is a c
    }
}
class MyClass6 implements M
{
    @Override
    public void show()
    {
        System.out.println("MyCla
    }
}
```

2. Supply the required information and click finish.



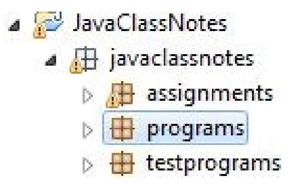
Creates folders corresponding to packages.

Source folder: JavaClassNotes

Name: javaclassnotes.programs

Create package-info.java

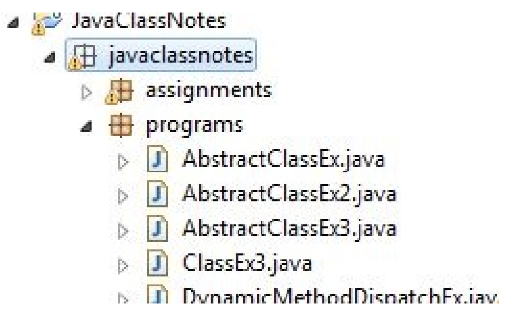
3. Now you’ll see the package in your package explorer view e.g. it may look like as below:



```
JavaClassNotes
    javaclassnotes
        assignments
        programs
        testprograms
```

[Note: here in this package, already there are some packages and classes-which we made earlier.]

4. Now Right click the package name->new->Class/Package etc to put classes/sub packages etc. inside the created package. It may have the following structure:



```
JavaClassNotes
  javaclassnotes
    assignments
    programs
      AbstractClassEx.java
      AbstractClassEx2.java
      AbstractClassEx3.java
      ClassEx3.java
      DynamicMethodDispatchEx.jav
```

[Note: here in this package, already there are some classes-which we made earlier.]

Implementation:

Now let us go through an example. Consider two travel companies-**a** and **b**. Company a conducts tours for Goa and Kerala. Company b conducts tours for Goa and Andaman. Any tourist can seek information from them for any particular tour package.

Here we have covered both of the following scenarios:

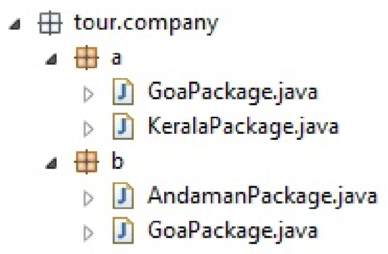
1. Less challenging situation:

Only Company **a** conducts tour for Kerala and Only company **b** conducts tour for Andaman.

2. More Challenging situation:

Notice that both of the companies are providing tour for Goa. And we need to get the information through the GoaPackage Class. [See both the packages are using the same class name]

Eclipse Package Explorer View:



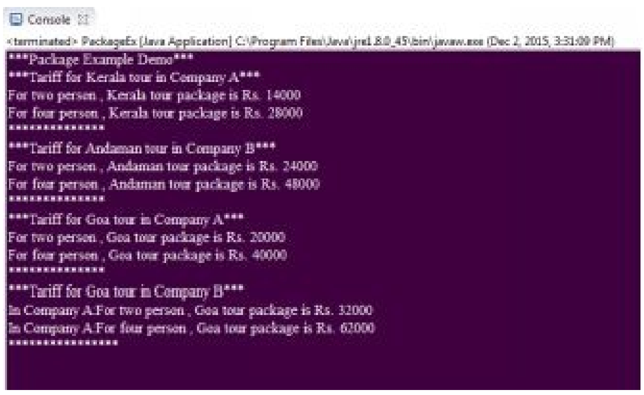
```
tour.company
  a
    GoaPackage.java
    KeralaPackage.java
  b
    AndamanPackage.java
    GoaPackage.java
```

```
// GoaPackage.java [For Company A, in tour.company.a package]
package tour.company.a;
public class GoaPackage
{
    int basic_price=10000;
    public void ShowPrice()
    {
            System.out.println("***Tariff for Goa tour in Company A***" );
            System.out.println("For two person , Goa tour package is Rs. "+ basic_price*2
);
            System.out.println("For four person , Goa tour package is Rs. "+
basic_price*4 );
                System.out.println("**************" );
    }
}

//  KeralaPackage.java [For Company A, in tour.company.a package]
package tour.company.a;

public class KeralaPackage
{
    int basic_price=7000;
    public void ShowPrice()
    {
                System.out.println("***Tariff for Kerala tour in Company A***" );
                System.out.println("For two person , Kerala tour package is Rs. "+
basic_price*2 );
                System.out.println("For four person , Kerala tour package is Rs. "+
basic_price*4 );
                System.out.println("**************" );
    }
}
// AndamanPackage.java [For Company B, in tour.company.b package]
package tour.company.b;
public class AndamanPackage
{
    int basic_price=12000;
    public void ShowTariff()
    {
                System.out.println("***Tariff for Andaman tour in Company B***" );
                System.out.println("For two person , Andaman tour package is Rs. "+
basic_price*2 );
                System.out.println("For four person , Andaman tour package is Rs. "+
basic_price*4 );
```

```
                System.out.println("**************" );
    }
}
// GoaPackage.java [For Company B, in tour.company.b package]
package tour.company.b;
public class GoaPackage
{
    int basic_price=15000;
    int serviceTax=2000;
    public void ShowTariff()
    {
                int forTwoPerson=basic_price*2 +serviceTax;
                int forFourPerson=basic_price*4 +serviceTax;
                System.out.println("***Tariff for Goa tour in Company B***" );
                System.out.println("In Company A:For two person , Goa tour package is Rs.
"+ forTwoPerson);
                System.out.println("In Company A:For four person , Goa tour package is Rs.
"+ forFourPerson );
                System.out.println("****************" );
    }
}
//Our main
package javaclassnotes.programs;
/*
import tour.company.a.KeralaPackage;
import tour.company.a.GoaPackage;
//or, simply use the following statement*/
import tour.company.a.*;
//For company B packages
import tour.company.b.*;

public class PackageEx {
    public static void main(String args[])
```

```
  {
        System.out.println("***Package Example Demo***");
        // Only CompanyA has KeralaPackage
        KeralaPackage keralaPackageInA=new KeralaPackage();
        keralaPackageInA.ShowPrice();
        //Only CompanyB has AndamanPackage
        AndamanPackage companyBAndamanPackage=new AndamanPackage();
        companyBAndamanPackage.ShowTariff();
        //Company A and B both have package for //Goa.
        tour.company.a.GoaPackage companyAGoaPackage=new
tour.company.a.GoaPackage();
        //GoaPackage companyAGoaPackage=new //GoaPackage();
        companyAGoaPackage.ShowPrice();
        tour.company.b.GoaPackage companyBGoaPackage=new
tour.company.b.GoaPackage();
        companyBGoaPackage.ShowTariff();
  }

}
```

Output:



```
<terminated> PackageEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Dec 2, 2015, 3:31:09 PM)
***Package Example Demo***
***Tariff for Kerala tour in Company A***
For two person , Kerala tour package is Rs. 14000
For four person , Kerala tour package is Rs. 28000
***************
***Tariff for Andaman tour in Company B***
For two person , Andaman tour package is Rs. 24000
For four person , Andaman tour package is Rs. 48000
***************
***Tariff for Goa tour in Company A***
For two person , Goa tour package is Rs. 20000
For four person , Goa tour package is Rs. 40000
***************
***Tariff for Goa tour in Company B***
In Company A:For two person , Goa tour package is Rs. 32000
In Company A:For four person , Goa tour package is Rs. 62000
*****************
```

Note:

1. Remember that in Java, all classes in java.lang package, are imported by default.

2. You must remember the visibility control mechanism with the following table:

	public	*protected*	*private*	*Default/No*

				modifier
Same class	*Yes*	*Yes*	*Yes*	*Yes*
Subclass in same package	*Yes*	*Yes*	*No*	*Yes*
Non-subclass in same package	*Yes*	*Yes*	*No*	*Yes*
Subclass in different package	*Yes*	*Yes*	*No*	*No*
Non-subclass in different package	*Yes*	*No*	*No*	*No*

OOPs Concepts Revisited

The fundamental features of object oriented programming is as below:

1. Class and Object
2. Polymorphism
3. Abstraction
4. Encapsulation
5. Inheritance
6. Message passing
7. Dynamic binding.

Teacher asks:

Can you tell me how we have covered these in Java?

1. Class and Object-Almost in every example, we have used classes and objects.
2. Polymorphism-Method overloading and overriding.
3. Abstraction-Abstract classes and Interfaces.
4. Encapsulation-Each class can be an example. A more effective example can be a class with a private member and getter-setter.
5. Inheritance-Examples in inheritance.
6. Message passing- Mostly observed in a multithreaded environment. But also the dynamic method dispatch example can be treated in this category.
7. Dynamic binding –Through the example in the dynamic method dispatch (method overriding).

Students ask:

What are the different types of polymorphism?

1. Compile time polymorphism-Which method needs to call is resolved by the compiler. So, it is also known as *early binding*. Since the call is resolved early, it is *faster* in general.
2. Run time polymorphism (Or, Dynamic Polymorphism) –Which method needs to call will be decided during runtime. That is why, it is also known as *late binding* and it is *slower* compared to early binding.

Students ask:

Does Java support pointers like C/C++?

No. One of the main reason is we can access beyond our intended data boundary which is really dangerous. Apart from this, if we support pointers, memory management will become tedious, because, in many cases, they are error-prone. We believe, as long as we are in the Java execution environment, we’ll never feel the need of using a pointer.

Use of static keyword

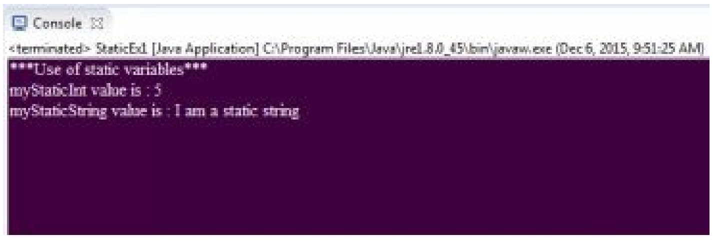
Sometimes we need variables that can be used without creating any object of that class. To serve that purpose, we tag the member/s with the keyword *static*. When a member is preceded with the keyword *static*, the member can be accessed before any object of that class is created i.e. we do not need to reference any object in this context.

Consider the below example:

Implementation-1:

```
package javaclassnotes.programs;


class StaticDemo1
{
   //static members
   static int myStaticInt=5;
   static String myStaticString="I am a static string";
   //Non static members
   int myNonStaticInt=25;
}
public class StaticEx1
{
   public static void main(String args[])
   {
            System.out.println("***Use of static variables***");
            //We can call static members with the class name itself
            //No need to create objects
            System.out.println("myStaticInt value is : "+StaticDemo1.myStaticInt);
            System.out.println("myStaticString value is : "+StaticDemo1.myStaticString);
            //Error:We cannot call instance variable with class name.
            //System.out.println("myNonStaticInt value is :
"+StaticDemo1.myNonStaticInt);


   }
}
```

Output:



```
<terminated> StaticEx1 [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Dec 6, 2015, 9:51:25 AM)
***Use of static variables***
myStaticInt value is : 5
myStaticString value is : I am a static string
```

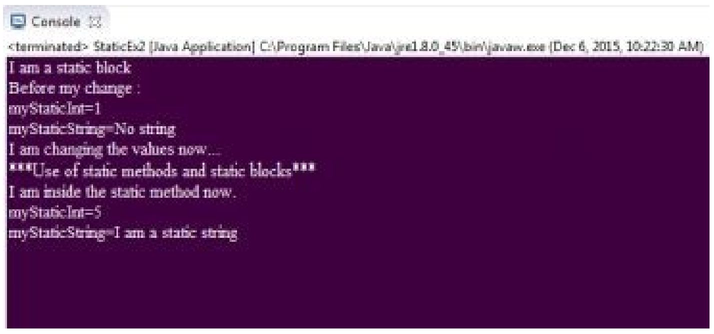
Analysis: We can see that we are accessing the static members with <classname>.<member Name> i.e. we do not need to create an object to access those variables.

Implementation-2:

Let us go through another example. Here we'll test static variables initialization with a static method and a static block.

```
package javaclassnotes.programs;

class StaticEx2

{

    //static members

    static int myStaticInt=1;

    static String myStaticString="No string";

    //instance variable

    int nonStaticInt=25;

    //static method

    static void setValuesToStaticMembers()

    {

                System.out.println("I am inside the static method now.");

                System.out.println("myStaticInt="+ myStaticInt);

                System.out.println("myStaticString="+ myStaticString);

    //error:Can access only static fields from here

                //System.out.println("myNonStaticInt="+ myNonStaticInt);

    }

    //static block

    static

    {

                System.out.println("I am a static block");

                System.out.println("Before my change :");
                System.out.println("myStaticInt="+ myStaticInt);
                System.out.println("myStaticString="+ myStaticString);
                System.out.println("I am changing the values now…");
                myStaticInt=5;
                myStaticString="I am a static string";
    }
    public static void main(String args[])
    {
                System.out.println("***Use of static methods and static blocks***");
                StaticEx2.setValuesToStaticMembers();
    }
}
```

Output:



Console

```
<terminated> StaticEx2 [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Dec 6, 2015, 10:22:30 AM)
I am a static block
Before my change :
myStaticInt=1
myStaticString=No string
I am changing the values now…
***Use of static methods and static blocks***
I am inside the static method now.
myStaticInt=5
myStaticString=I am a static string
```

Analysis:

Look at the output carefully. Look at the order of the output. *You can see the statements in the static block printed on the top of output. Even before the execution of the static block, the static variables were initialized. Later static block changed the values (which are reflected clearly when we call the static method).*

It is because as soon as a static class loaded, all static statements run.

You should notice another important characteristic also: Static methods can access only static members.

Also note that our static method is *nested* here.

Rules of Thumb:

1. Static methods can only other static methods.

2. Static methods can access only static fields.

3. Static methods cannot refer this or super.

Students ask:

Can we create static class?

Yes. But there is a constraint. The static class should be inside of another class i.e. it must be nested. *Java does not allow us to create top level static class.*

The class which contains the static class is termed as an outer class.

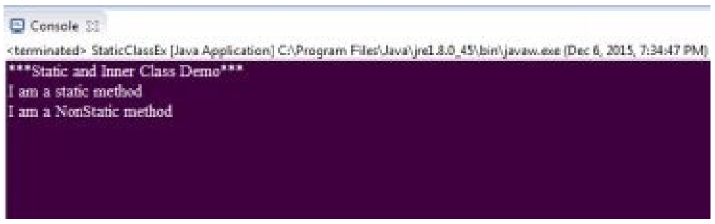
Consider the below example. *Here we have shown how to create and use a nested static class and a nested non-static (Inner class).*

Implementation-3:

```
package javaclassnotes.programs;
//Java doesn’t allow us to create top-level static classes,it must be  nested.
class OuterClass
{
    //static class
    static class MyStaticClass
    {
            public static void showStaticMethod()
            {
                    System.out.println(“I am a static method”);
            }
    }
    //non static inner class
    public class MyNonStaticClass
    {
            public void showNonStaticMethod()
            {
                    System.out.println(“I am a NonStatic method”);
            }
    }
}
```

```
class StaticClassEx
{
    public static void main(String args[])
    {
                System.out.println("***Static and Inner Class Demo***");
                //Call Static method
OuterClass.MyStaticClass.showStaticMethod();
                //CallNonStatic method
                OuterClass.MyNonStaticClass obNonStatic=(new OuterClass()).new
MyNonStaticClass();
                obNonStatic.showNonStaticMethod();
    }
}
```

Output:



```
Console
<terminated> StaticClassEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Dec 6, 2015, 7:34:47 PM)
***Static and Inner Class Demo***
I am a static method
I am a NonStatic method
```

Students ask:

What is an inner class?

As described above, a non-static nested class is termed as an Inner class. It can access all the variables and methods of the outer class.

Students ask:

From static class, can we access the variables of Outer class?

Answer is yes if and only if those variables are static. Consider the below example. Following code snippet is fine:

```
class OuterClass
{
    static int outer_int=25;
    //static class
    static class MyStaticClass
    {
            public static void showStaticMethod()
            {
                    System.out.println("I am a static method");
                    System.out.println("Outer_int =" +outer_int);
            }
    }
}
```

But if outer_int is non-static, compiler will raise an issue.

Assignment:

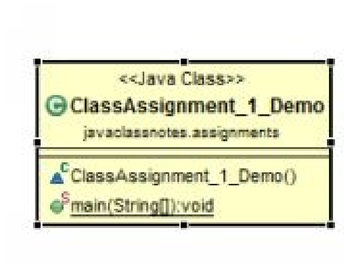
1. Suppose you have formed a cricket team. Now your team is going to play against an opponent team. You must be aware of the fact that which team will bat (or bowl) first will be decided through the toss and you need to send your captain for that. So, at first, you must elect a captain. At the same time, you must be aware that you can select one and only one captain. So, if you do not have any such captain, you will select one and send him for toss. Otherwise, you simply send the already nominated captain for the toss. Can you design this?

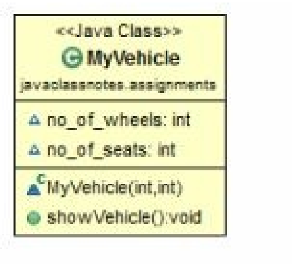
Solution to the Assignments

Class

1. Create a class Vehicle. The class should have two fields-no_of_seats and no_of_wheels. Create two objects-Motorcycle and Car for this class. Your output should show the descriptions for Car and Motorcycle.

Uml Class Diagram:





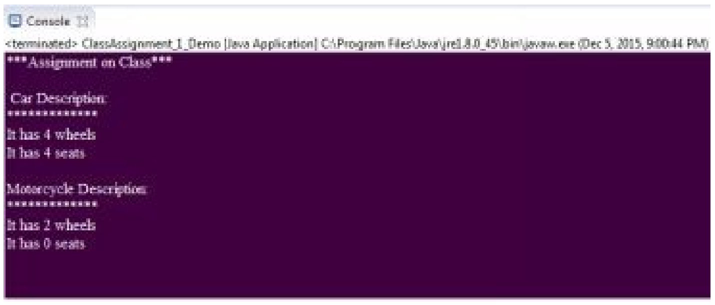
Implementation:

[No need to include package statements. But once you understand the concept of package, we'll see that it is very much handy to make an organized structure for our programs.]

```
package javaclassnotes.assignments;
class MyVehicle
{
    int no_of_wheels;
    int no_of_seats;
    MyVehicle(int wheels,int seats)
    {
            no_of_wheels=wheels;
            no_of_seats=seats;
    }
    public void showVehicle()
    {
            System.out.print("Description:");
            System.out.println("\n************");
            System.out.println("It has "+ no_of_wheels+" wheels");
            System.out.println("It has "+ no_of_seats+" seats\n");
    }
}
```

```
class ClassAssignment_1_Demo
{
    public static void main(String args[])
    {
            System.out.print(“***Assignment on Class***\n\n “);
            MyVehicle car=new MyVehicle(4,4);
            MyVehicle motorCycle=new MyVehicle(2,0);
            System.out.print(“Car “);
            car.showVehicle();
            System.out.print(“Motorcycle “);
            motorCycle.showVehicle();
    }

}
```

Output:

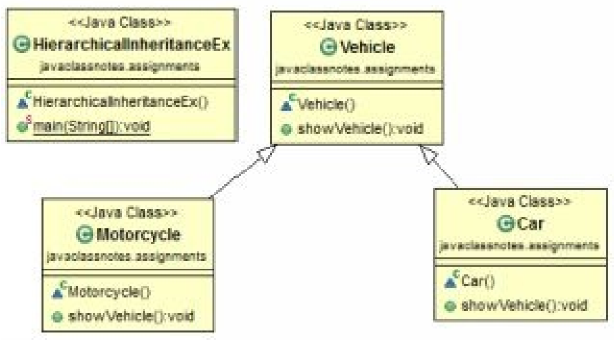


```
<terminated> ClassAssignment_1_Demo [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Dec 5, 2015, 9:00:44 PM)
***Assignment on Class***

 Car Description:
**************
It has 4 wheels
It has 4 seats

Motorcycle Description:
**************
It has 2 wheels
It has 0 seats
```

Inheritance

1. Write a Simple Program to implement Hierarchical Inheritance.

Uml Class Diagram:

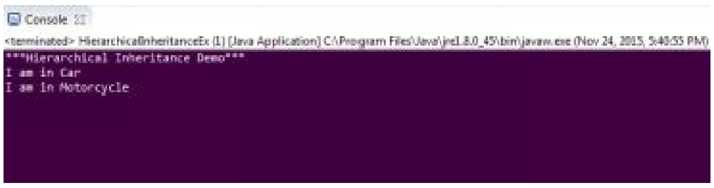


Implementation:

```
package javaclassnotes.assignments;

class Vehicle
{
   public void showVehicle()
   {
            System.out.println("I am in Vehicle");
   }
}
class Car extends Vehicle
{
   public void showVehicle()
   {
            System.out.println("I am in Car");
   }
}
class Motorcycle extends Vehicle
{
   public void showVehicle()
   {
            System.out.println("I am in Motorcycle");
   }
}

class HierarchicalInheritanceEx
{
   public static void main(String args[])
   {
            System.out.println("***Hierarchical Inheritance Demo***");
            Car c=new Car();
            c.showVehicle();
```

```
                Motorcycle m=new Motorcycle();
                m.showVehicle();
    }
}
```

Output:

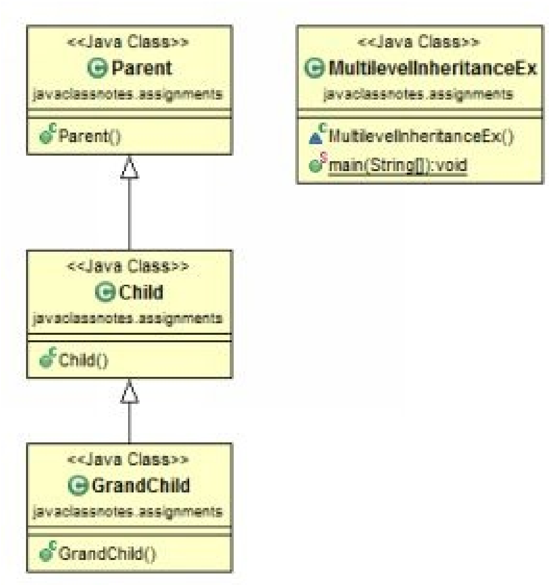


```
<terminated> HierarchicalInheritanceEx (1) [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 24, 2015, 5:40:55 PM)
***Hierarchical Inheritance Demo***
I am in Car
I am in Motorcycle
```

2. Write a Simple Program to implement Multilevel Inheritance.

We could write the program like the above program (See our code structure in the chapter on Inheritance).Just for a variety, we are using only constructors here. If you want to see the concrete methods implementation, just uncomment the codes in the below program.

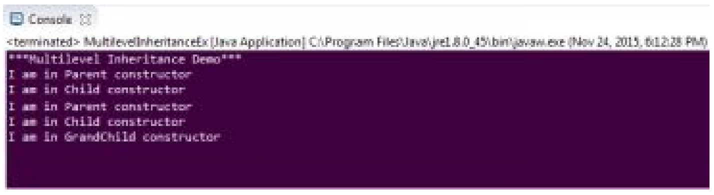
Uml Class Diagram:



Implementation:

```
package javaclassnotes.assignments;


class Parent
{
    public Parent()
    {
            System.out.println("I am in Parent constructor");
    }
    /*public void showMe()
    {
            System.out.println("I am a Parent");
    }        */
}
class Child extends Parent
{
    public Child()
    {
            System.out.println(“I am in Child constructor”);
    }
    /*public void showMe()
    {
            System.out.println(“I am a Child”);
    }        */
}
class GrandChild extends Child
{
    public GrandChild()
    {
            System.out.println(“I am in GrandChild constructor”);
    }
    /*public void showMe()
    {
            System.out.println(“I am a GrandChild”);
    }*/
}

class MultilevelInheritanceEx
{
    public static void main(String args[])
    {
            System.out.println(“***Multilevel Inheritance Demo***”);
            //Parent p=new Parent();
            //p.showMe();
            Child c=new Child();
            //c.showMe();
            GrandChild g=new GrandChild();
            //g.showMe();
    }
}
```

Output:



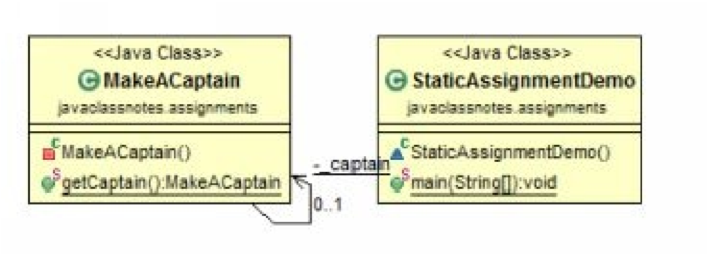
```
Console
<terminated> MultilevelInheritanceEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 24, 2015, 6:12:28 PM)
***Multilevel Inheritance Demo***
I am in Parent constructor
I am in Child constructor
I am in Parent constructor
I am in Child constructor
I am in GrandChild constructor
```

Note the output. Whenever we want to initiate a child class object, the parent class constructors called automatically first. This is why, Child class constructor called Parent class constructor first. Similarly, our GrandChild class constructor called its parent (i.e. Child class constructor) which in turn called its parent (i.e. Parent class constructor) constructor first.

Use of static keyword

1. Suppose you have formed a cricket team. Now your team is going to play against an opponent team. You must be aware of the fact that which team will bat (or bowl) first will be decided through the toss and you need to send your captain for that. So, at first, you must elect a captain. At the same time, you must be aware that you can select one and only one captain. So, if you do not have any such captain, you will select one and send him for toss. Otherwise, you simply send the already nominated captain for the toss. Can you design this?

Uml Class Diagram:

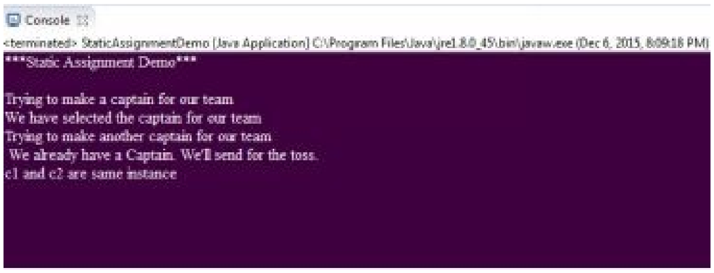


Implementation:

package javaclassnotes.assignments;

```
class NominateACaptain
{
    private static NominateACaptain _captain;
    //We make the constructor private to prevent the use of "new"
    private NominateACaptain() { }
    // public static synchronized MakeACaptain getCaptain()
    public static  NominateACaptain getCaptain()
    {
                // Lazy initialization
                if (_captain == null)
                { _captain = new NominateACaptain();
                System.out.println(“We have selected the captain for our team”);
                }
                else
                {
                            System.out.print(” We already have a Captain.”);
                            System.out.println(” We’ll send for the toss.”);
                }
                return _captain;

    }
}

class StaticAssignmentDemo
{
    public static void main(String[] args)
    {
                System.out.println(“***Static Assignment Demo***\n”);
                System.out.println(“Trying to make a captain for our team”);
                NominateACaptain c1 = NominateACaptain.getCaptain();
                System.out.println(“Trying to make another captain for our team”);
                NominateACaptain c2 = NominateACaptain.getCaptain();
                if (c1 == c2)
                {
                            System.out.println(“c1 and c2 are same instance”);
                }

    }
}
```

Output:



```
<terminated> StaticAssignmentDemo [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Dec 6, 2015, 8:09:18 PM)
***Static Assignment Demo***

Trying to make a captain for our team
We have selected the captain for our team
Trying to make another captain for our team
 We already have a Captain. We'll send for the toss.
c1 and c2 are same instance
```

FAQ

Now it is the time to test your understanding .Please go through the questions. If there is any doubt, please go back to the respective topic.

1. What is a class?

2. What is an object?

3. Differentiate between object and reference?

4. Can we implement multiple inheritance in Java?

5. Can we implement hybrid inheritance in Java?

6. Differentiate between an abstract class and an interface.

7. Differentiate between method overloading and method overriding.

8. How you can implement dynamic polymorphism in Java?

9.”Package statement should always come on top”-is it true?

10. What is JVM?

11. Differentiate between JRE and JDK.

12. What is an inner class?

13. How you can create a static class in java?

14. How you can implement abstraction and encapsulation in Java?

15. Differentiate between a static binding and a dynamic binding in Java.

16. What is use of “super” in Java?

17. What is the use of “this” in Java?

18. What is use of “default” in Java?

19. Can you an abstract class without an abstract method?

20. Can you inherit constructors?

21. What is the use of “final” in Java?

22. Differentiate between an instance method and a class method (static method)?

23. Can you create a static block? What is its use?

24. What is the default package in Java?

25. Does java support pointers?

Reference

http://www.javatpoint.com/

http://www.tutorialspoint.com/

http://beginnersbook.com/

https://docs.oracle.com/javase/tutorial

http://freefeast.info/difference-between/difference-between-runtime-polymorphism-and-compile-time-polymorphism/

http://www.javabeat.net/what-is-the-difference-between-jrejvm-and-jdk/

http://www.geeksforgeeks.org/static-class-in-java/

http://javarevisited.blogspot.in/2015/04/3-ways-to-prevent-method-overriding-in.html

http://mindprod.com/jgloss/interfacevsabstract.html

http://www.programmerinterview.com/index.php/java-questions/interface-vs-abstract-class/

http://www.javaworld.com/article/2077421/learn-java/abstract-classes-vs-interfaces.html

Acknowledgements

My sincere thanks to my family, my friends, my great teachers and to all of them who supported this project directly or indirectly. Though it is my book but I believe that with the help of these extraordinary people only, I was able to complete this work. Again thanks to all of them who helped me to fulfil this project to motivate others in Java.

About the Author

Vaskaran Sarcar (ME (Software Engineering), MCA, B Sc. (Math)) is a Senior Software Engineer at Hewlett Packard India Software Operation Pvt. Ltd. He is working at the HP India PPS R&D division since August, 2009. He is also the author of the books- *Design Patterns in C#, Design Patterns in JAVA, Java Design Patterns (A revised and enhanced version-coming in December,2015), Operating System: Computer Science Interview Series, Easy manifestation and Sweetheart, You Can Also Manifest Your Goal*. He devoted his early years (2005-2007) in teaching in various engineering colleges. Later he got MHRD-GATE Scholarship (India) from 2007-2009.Reading and learning new things are passion for him. You can catch him for any comments, suggestions or further improvements at: vaskaran@rediffmail.com.

###