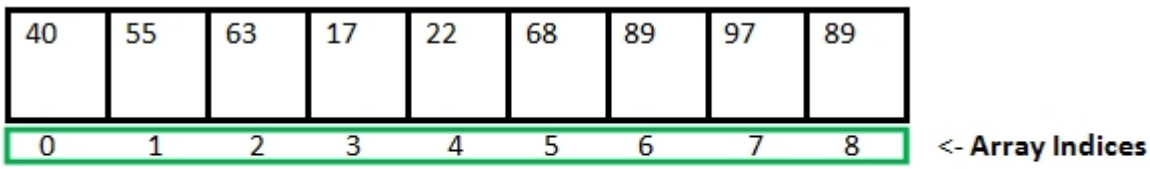
UNIT II

Arrays in Java

An array is a group of like-typed variables that are referred to by a common name.Arrays in Java work differently than they do in C/C++. Following are some important point about Java arrays.

- In Java all arrays are dynamically allocated.(discussed below)
- Since arrays are objects in Java, we can find their length using member length. This is different from C/C++ where we find length using sizeof.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered and each have an index beginning from 0.
- Java array can be also be used as a static field, a local variable or a method parameter.
- The **size** of an array must be specified by an int value and not long or short.
- The direct superclass of an array type is Object.
- Every array type implements the interfaces Cloneable and java.io.Serializable.

Array can contains primitives (int, char, etc) as well as object (or non-primitives) references of a class depending on the definition of array. In case of primitives data types, the actual values are stored in contiguous memory locations. In case of objects of a class, the actual objects are stored in heap segment.



Creating, Initializing, and Accessing an Array

One-Dimensional Arrays :

The general form of a one-dimensional array declaration is

```
type var-name[];
OR
type[] var-name;
```

An array declaration has two components: the type and the name. *type* declares the element type of the array. The element type determines the data type of each element that comprises the array. Like array of int type, we can also create an array of other primitive data types like char, float, double..etc or user defined data type(objects of a class).Thus, the element type for the array determines what type of data the array will hold.

Instantiating an Array in Java

When an array is declared, only a reference of array is created. To actually create or give memory to array, you create an array like this:The general form of *new* as it applies to one-dimensional arrays appears as follows:

```
var-name = new type [size];
```

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *var-name* is the name of array variable that is linked to the array. That is, to use *new* to allocate an array, **you must specify the type and number of elements to allocate.**

Example:

```
int intArray[]; //declaring array
intArray = new int[20]; // allocating memory to array
```

OR

```
int[] intArray = new int[20]; // combining both statements in one
```

Java Multidimensional Arrays

In this tutorial, we will learn about the Java multidimensional array using 2-dimensional arrays and 3-dimensional arrays with the help of examples.

Before we learn about the multidimensional array, make sure you know about Java array.

A multidimensional array is an array of arrays. Each element of a multidimensional array is an array itself. For example,

```
int[][] a = new int[3][4];
```

Here, we have created a multidimensional array named a. It is a 2-dimensional array, that can hold a maximum of 12 elements,

	Column 1	Column 2	Column 3	Column 4
Row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Remember, Java uses zero-based indexing, that is, indexing of arrays in Java starts with 0 and not 1.

Let's take another example of the multidimensional array. This time we will be creating a 3-dimensional array. For example,

```
String[][][] data = new String[3][4][2];
```

Here, `data` is a 3d array that can hold a maximum of 24 (3*4*2) elements of type `String`.

How to initialize a 3d array in Java?

Let's see how we can use a 3d array in Java. We can initialize a 3d array similar to the 2d array. For example,

```
// test is a 3d array
int[][][] test = {
      {
       {1, -2, 3},
       {2, 3, 4}
      },
      {
       {-4, -5, 6, 9},
       {1},
       {2, 3}
      }
};
```

Basically, a 3d array is an array of 2d arrays. The rows of a 3d array can also vary in length just like in a 2d array.

Example: 3-dimensional Array

```
class ThreeArray {
   public static void main(String[] args) {
      // create a 3d array
      int[][][] test = {
         {
          {1, -2, 3},
          {2, 3, 4}
         },
         {
          {-4, -5, 6, 9},
          {1},
```

```
            {2, 3}
        }       };
    // for..each loop to iterate through elements of 3d array
    for (int[][] array2D: test) {
        for (int[] array1D: array2D) {
            for(int item: array1D) {
                System.out.println(item);
            }    }    }    }    }
```

Output:

```
1
-2
3
2
3
4
-4
-5
6
9
1
2
3
```

Inheritance in Java Programming with examples

BY CHAITANYASINGH | FILED UNDER: OOPS CONCEPT

The process by which one class acquires the properties(data members) and functionalities(methods) of another class is called **inheritance**. The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.

Child Class:

The class that extends the features of another class is known as child class, sub class or derived class.

ParentClass:

The class whose properties and functionalities are used(inherited) by another class is known as parent class, super class or Base class.

Inheritance is a process of defining a new class based on an existing class by extending its common data members and methods.Inheritance allows us to reuse of code, it improves reusability in your java application.Note: The biggest **advantage of Inheritance** is that the code that is already present in base class need not be rewritten in the child class.

This means that the data members(instance variables) and methods of the parent class can be used in the child class as.

If you are finding it difficult to understand what is class and object then refer the guide that I have shared on object oriented programming: OOPs Concepts

Syntax: Inheritance in Java

To inherit a class we use extends keyword. Here class XYZ is child class and class ABC is parent class. The class XYZ is inheriting the properties and methods of ABC class.

```
class XYZ extends ABC
{
}
```

Inheritance Example

In this example, we have a base class `Teacher` and a sub class `PhysicsTeacher`. Since class `PhysicsTeacher` extends the designation and college properties and work() method from base class, we need not to declare these properties and method in sub class.
Here we have collegeName, designation and work() method which are common to all the teachers so we have declared them in the base class, this way the child classes like `MathTeacher`, `MusicTeacher` and `PhysicsTeacher` do not need to write this code and can be used directly from base class.

```
class Teacher {
  String designation = "Teacher";
  String collegeName = "Beginnersbook";
  void does(){
            System.out.println("Teaching");
  }
}
public class PhysicsTeacher extends Teacher{
  String mainSubject = "Physics";
  public static void main(String args[]){
            PhysicsTeacher obj = new PhysicsTeacher();
            System.out.println(obj.collegeName);
            System.out.println(obj.designation);
            System.out.println(obj.mainSubject);
            obj.does();
  } }
```

Output:

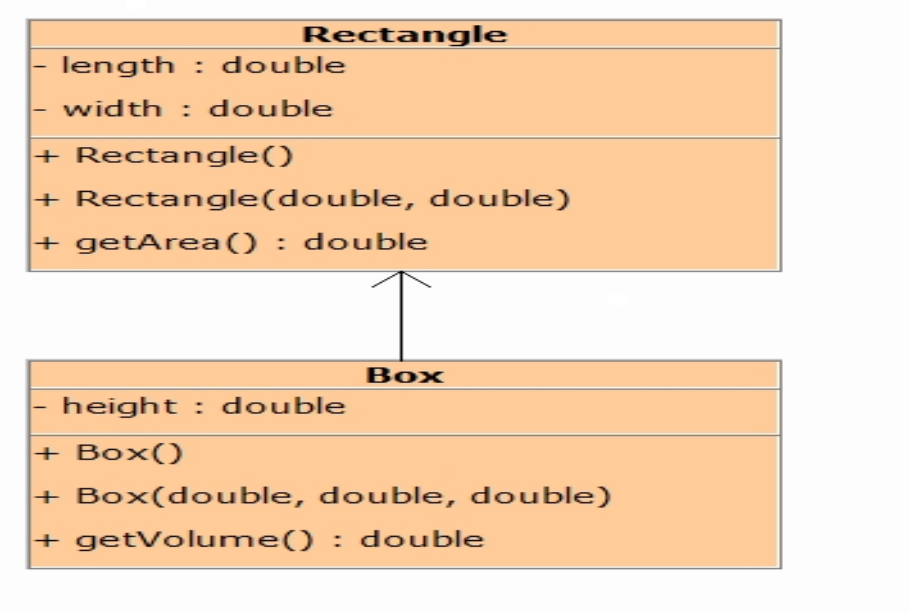
```
Beginnersbook
Teacher
Physics
Teaching
```

Call the Super class Constructor

A subclass can have its own private data members, so a subclass can also have its own constructors.

The constructors of the subclass can initialize only the instance variables of the subclass. Thus, when a subclass object is instantiated the subclass object must also automatically execute one of the constructors of the superclass.

To call a superclass constructor the **super** keyword is used. The following example programs demonstrate use of super keyword.



(Rectangle.java)

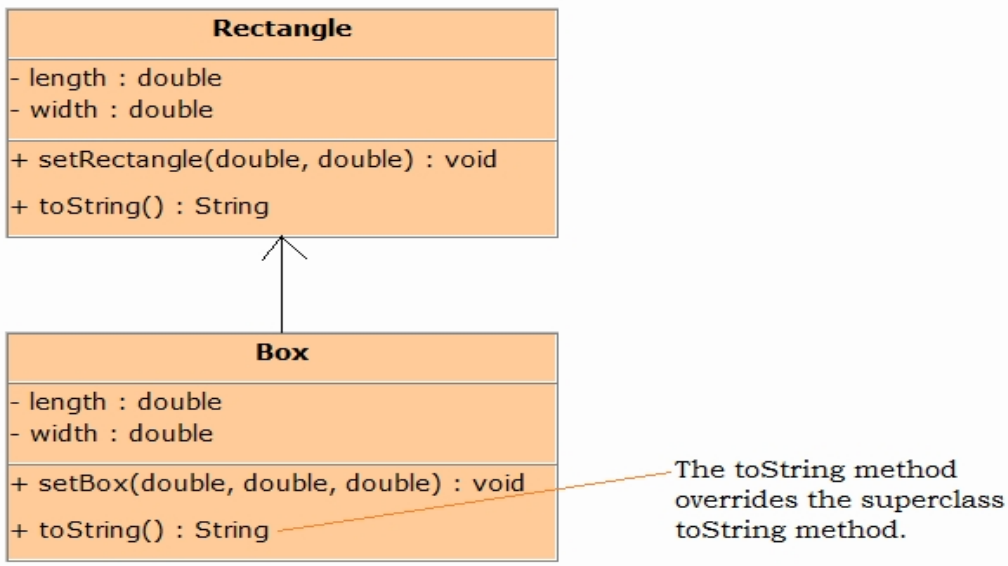
```
/**
 * This class holds data of a Rectangle.
 */
public class Rectangle
{
  private double length;  // To hold length of rectangle
  private double width;  // To hold width of rectangle
  /**
   * The constructor initialize rectangle's
   * length and width with default value
   */
  public Rectangle()
  {
    length = 0;
    width = 0;
  }
  /**
   * The constructor accepts the rectangle's
   * length and width.
   */
```

```
  public Rectangle(double length, double width)
  {
    this.length = length;
    this.width = width;
  }
  /**
   * The getArea method returns the area of
   * the rectangle.
   */
  public double getArea()
  {
    return length * width;
  }
}
```

Overriding Super class Methods

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

A subclass may call an overridden superclass method by prefixing its name with the super key word and a dot (.). Consider the following:



Program (BoxDemo.java)

```
import java.util.Scanner;

public class BoxDemo

{
```

```
  public static void main(String[] args)
  {
    double length, width, height;
    // Create a Scanner object for keybaord input.
    Scanner console = new Scanner(System.in);
    // Get the length of box.
    System.out.print( "Enter the length of box : ");
    length = console.nextDouble();
    // Get the width of box.
    System.out.print( "Enter the width of box : ");
    width = console.nextDouble();
    // Get the height of box.
    System.out.print( "Enter the height of box : ");
    height = console.nextDouble();
    // Create a box object.
    Box myBox = new Box();
    // Set the length, width and height of box.
    myBox.setBox(length,  width, height);
    // Display the box detalis.
    System.out.println(myBox);
  }
}
```

Output :

```
Enter the length of box : 34.5
Enter the width of box : 5.6
Enter the height of box : 3.2
Length : 34.5
Width : 5.6
Height : 3.2
```

Java Inheritance (Subclass and Superclass)

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- **subclass** (child) - the class that inherits from another class

- **superclass** (parent) - the class being inherited from

To inherit from a class, use the `extends` keyword.

In the example below, the `Car` class (subclass) inherits the attributes and methods from the `Vehicle` class (superclass):

Example

```
class Vehicle {
  protected String brand = "Ford";        // Vehicle attribute
  public void honk() {                    // Vehicle method
    System.out.println("Tuut, tuut!");
  }
}
class Car extends Vehicle {
  private String modelName = "Mustang";    // Car attribute
  public static void main(String[] args) {


    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (from the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand attribute (from the Vehicle class) and the value of the modelName from the
Car class
    System.out.println(myCar.brand + " " + myCar.modelName);
  }
}
```

Did you notice the `protected` modifier in Vehicle?

We set the **brand** attribute in **Vehicle** to a `protected` access modifier. If it was set to `private`, the Car class would not be able to access it.

Why And When To Use "Inheritance"?

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

Tip: Also take a look at the next chapter, Polymorphism, which uses inherited methods to perform different tasks.

The final Keyword

If you don't want other classes to inherit from a class, use the `final` keyword:

If you try to access a `final` class, Java will generate an error:

```
final class Vehicle {
  ...
}

class Car extends Vehicle {
  ...
}
```

The output will be something like this:

```
Car.java:8: error: cannot inherit from final Vehicle
class Car extends Vehicle {
                  ^
1 error)
```

Polymorphism in Java

Polymorphism in Java is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

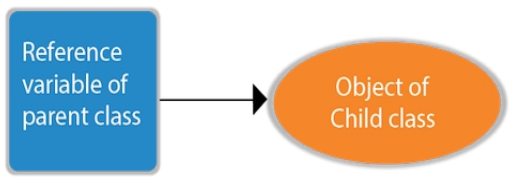
If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

Runtime Polymorphism in Java

Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:



Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
1.  class Bike{
2.    void run(){System.out.println( "running");}
3.  }
```

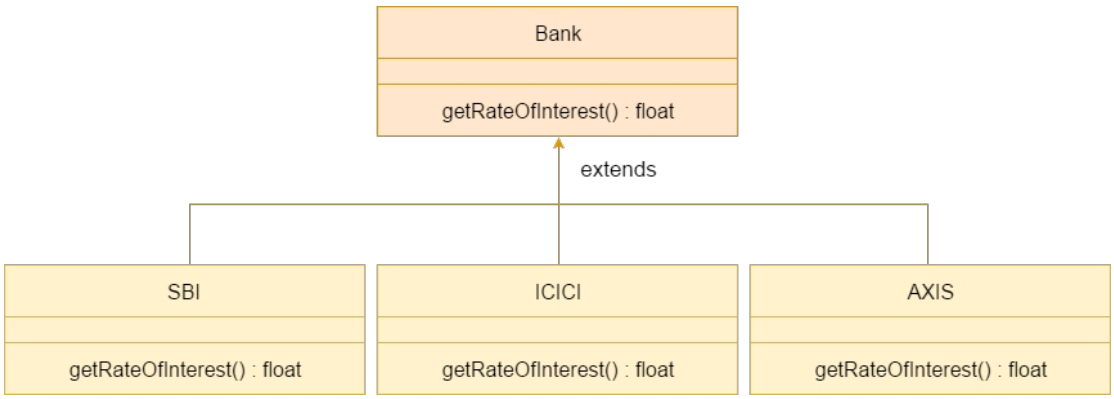
```
class Splendor extends Bike{
  void run(){System.out.println( "running safely with 60km" );}

  public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run();
  }
}
```

Output:

```
running safely with 60km.
```

Java Runtime Polymorphism Example: Bank

Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.




```
class Bank{
float getRateOfInterest(){ return 0;}
}
class SBI extends Bank{
float getRateOfInterest(){ return 8.4f;}
}
class ICICI extends Bank{
float getRateOfInterest(){ return 7.3f;}
}
class AXIS extends Bank{
float getRateOfInterest(){ return 9.7f;}
}
class TestPolymorphism{
public static void main(String args[]){
Bank b;
b=new SBI();
```

```
17. System.out.println( "SBI Rate of Interest: " +b.getRateOfInterest());
18. b=new ICICI();
19. System.out.println( "ICICI Rate of Interest: " +b.getRateOfInterest());
20. b=new AXIS();
21. System.out.println( "AXIS Rate of Interest: " +b.getRateOfInterest());
22. } }
```

Output:

```
SBI Rate of Interest: 8.4
ICICI Rate of Interest: 7.3
AXIS Rate of Interest: 9.7
```

Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Before learning the Java abstract class, let's understand the abstraction in Java first.

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

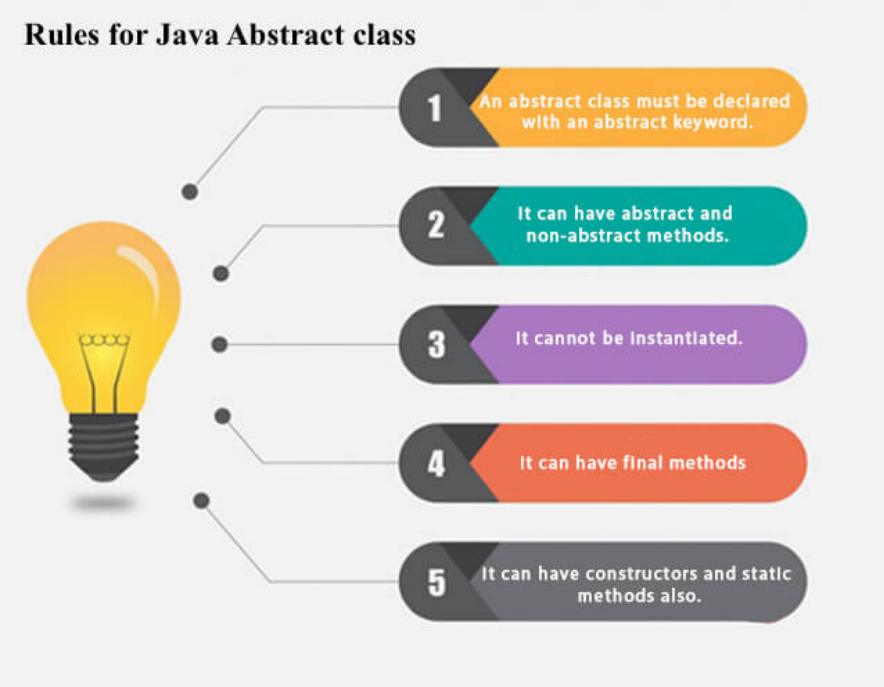
1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.



Interface in Java

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have **default and static methods** in an interface.

Since Java 9, we can have **private methods** in an interface.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

How to declare an interface?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

```
1. interface <interface_name>{
2. }
```

Fields

A *field* is a class, interface, or enum with an associated value. Methods in the `java.lang.reflect.Field` class can retrieve information about the field, such as its name, type, modifiers, and annotations. (The section Examining Class Modifiers and Types in the Classes lesson describes how to retrieve annotations.) There are also methods which enable dynamic access and modification of the value of the field. These tasks are covered in the following sections:

- Obtaining Field Types describes how to get the declared and generic types of a field
- Retrieving and Parsing Field Modifiers shows how to get portions of the field declaration such as public or transient
- Getting and Setting Field Values illustrates how to access field values
- Troubleshooting describes some common coding errors which may cause confusion

When writing an application such as a class browser, it might be useful to find out which fields belong to a particular class. A class's fields are identified by invoking

`Class.getFields()`. The `getFields()` method returns an array of `Field` objects containing one object per accessible public field.

A public field is accessible if it is a member of either:

- this class
- a superclass of this class
- an interface implemented by this class
- an interface extended from an interface implemented by this class

A field may be a class (instance) field, such as `java.io.Reader.lock` , a static field, such as `java.lang.Integer.MAX_VALUE` , or an enum constant, such as `java.lang.Thread.State.WAITING`.

Java and Multiple Inheritance

Multiple Inheritance is a feature of object oriented concept, where a class can inherit properties of more than one parent class. The problem occurs when there exist methods with same signature in both the super classes and subclass. On calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority.

Why Java doesn't support Multiple Inheritance?

Consider the below Java code. It shows error.

filter_none

edit

play_arrow

brightness_4

```
// First Parent class

class Parent1

{

   void fun()

   {

      System.out.println("Parent1");

   }
```

```
}

// Second Parent Class

class Parent2

{

   void fun()

   {

      System.out.println("Parent2");

   }

}

// Error : Test is inheriting from multiple

// classes

class Test extends Parent1, Parent2

{

  public static void main(String args[])

  {

     Test t = new Test();

     t.fun();

  }

}
```

Output :

Compiler Error

From the code, we see that, on calling the method fun() using Test object will cause complications such as whether to call Parent1's fun() or Parent2's fun() method.

1. **The Diamond Problem:**
   The below Java program throws compiler error when run. Multiple inheritance causes diamond problem when allowed in other languages like C++.

```
GrandParent
         /    \
        /      \
     Parent1      Parent2
        \      /
         \    /
           Test
```

filter_none
edit
play_arrow
brightness_4

```
// A Grand parent class in diamond
class GrandParent
{
    void fun()
    {
        System.out.println("Grandparent");
    }
}
  // First Parent class
class Parent1 extends GrandParent
{
    void fun()
    {
        System.out.println("Parent1");
```

```
    }
}
// Second Parent Class
class Parent2 extends GrandParent
{
    void fun()
    {
        System.out.println("Parent2");
    }
}
// Error : Test is inheriting from multiple
// classes
class Test extends Parent1, Parent2
{
   public static void main(String args[])
   {
       Test t = new Test();
       t.fun();
   }
}
```

Output :

Error :

```
prog.java:31: error: '{' expected
class Test extends Parent1, Parent2
                   ^
1 error
```

From the code, we see that: On calling the method fun() using Test object will cause complications such as whether to call Parent1's fun() or Child's fun() method.

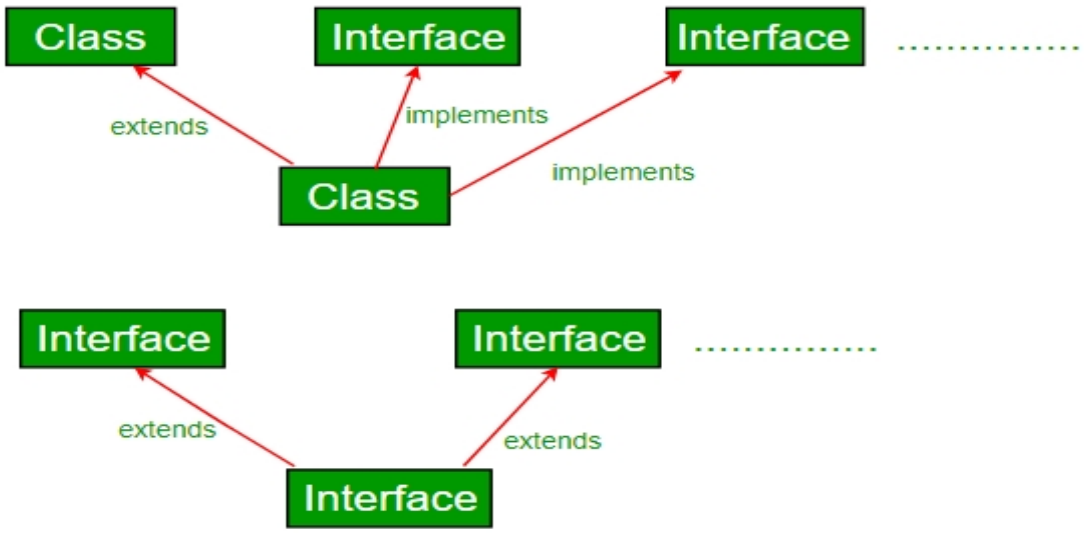
Therefore, in order to avoid such complications Java does not support multiple inheritance of classes.

2. Simplicity – Multiple inheritance is not supported by Java using classes , handling the complexity that causes due to multiple inheritance is very complex. It creates problem during various operations like casting, constructor chaining etc and the above all reason is that there are very few scenarios on which we actually need multiple inheritance, so better to omit it for keeping the things simple and straightforward

Interfaces and Inheritance in Java

Prerequisites: Interfaces in Java, Java and Multiple Inheritance

A class can extends another class and/ can implement one and more than one interface.



filter_none
edit

play_arrow

brightness_4

```
// Java program to demonstrate that a class can
```

```
// implement multiple interfaces
import java.io.*;
interface intfA
{
    void m1();
}
interface intfB
{
    void m2();
}
// class implements both interfaces
// and provides implementation to the method.
class sample implements intfA, intfB
{
    @Override
    public void m1()
    {
        System.out.println("Welcome: inside the method m1");
    }
      @Override
    public void m2()
    {
```

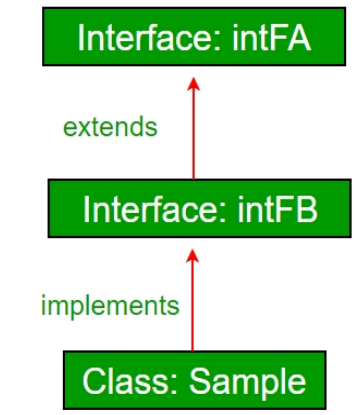
```
        System.out.println("Welcome: inside the method m2");
    }
}
 class GFG
{
    public static void main (String[] args)
    {
        sample ob1 = new sample();
        // calling the method implemented
        // within the class.
        ob1.m1();
        ob1.m2();
    }
}
```

Output;

Welcome: inside the method m1

Welcome: inside the method m2

Interface inheritance : An Interface can extend other interface.



Packages & Interfaces in Java

Introduction to Packages

A package is a mechanism to group the similar type of classes, interfaces and sub-packages and provide access control. It organizes classes into single unit.
In Java already many predefined packages are available, used while programming.
For example: java.lang, java.io, java.util etc.

Advantages of Packages

- Packages provide code reusability, because a package has group of classes.
- It helps in resolving naming collision when multiple packages have classes with the same name.
- Package also provides the hiding of class facility. Thus other programs cannot use the classes from hidden package.
- Access limitation can be applied with the help of packages.
- One package can be defined in another package.

Types of Packages

There are two types of packages available in Java.

1. Built-in packages

Built-in packages are already defined in java API. For example: java.util, java.io, java,lang, java.awt, java.applet, java.net, etc.

2. User defined packages

The package we create according to our need is called user defined package.

Creating a Package

We can create our own package by creating our own classes and interfaces together. The package statement should be declared at the beginning of the program.

Syntax:

```
package <packagename>;
class ClassName
{
……..
```

……..

}

Example: Creating a Package

```
// Demo.java
package p1;
class Demo
{
  public void m1()
  {
     System.out.println("Method m1..");
  }
}
```

Creating a Package

To create a package, you choose a name for the package (naming conventions are discussed in the next section) and put a `package` statement with that name at the top of *every source file* that contains the types (classes, interfaces, enumerations, and annotation types) that you want to include in the package.

The package statement (for example, `package graphics;`) must be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

Note: If you put multiple types in a single source file, only one can be public, and it must have the same name as the source file. For example, you can define public class Circle in the file Circle.java, define public interface Draggable in the file Draggable.java, define public enum Day in the file Day.java, and so forth.

You can include non-public types in the same file as a public type (this is strongly discouraged, unless the non-public types are small and closely related to the public type), but only the public type will be accessible from outside of the package. All the top-level, non-public types will be *package private*.

If you put the graphics interface and classes listed in the preceding section in a package called `graphics`, you would need six source files, like this:

```
//in the Draggable.java file
package graphics;
public interface Draggable {
   . . .
}

//in the Graphic.java file
package graphics;
```

```
public abstract class Graphic {
   . . .
}

//in the Circle.java file
package graphics;
public class Circle extends Graphic
   implements Draggable {
   . . .
}

//in the Rectangle.java file
package graphics;
public class Rectangle extends Graphic
   implements Draggable {
   . . .
}

//in the Point.java file
package graphics;
public class Point extends Graphic
   implements Draggable {
   . . .
}

//in the Line.java file
package graphics;
public class Line extends Graphic
   implements Draggable {
   . . .
}
```

If you do not use a `package` statement, your type ends up in an unnamed package. Generally speaking, an unnamed package is only for small or temporary applications or when you are just beginning the development process. Otherwise, classes and interfaces belong in named packages.

Packages in Java: How to Create and Use Packages in Java?

One of the most innovative features of Java is the concept of packages. Packages in Java are a way to encapsulate a group of classes, interfaces, enumerations, annotations, and sub-packages. Conceptually, you can think of java packages as being similar to different folders on your computer. In this tutorial, we will cover the basics of packages in Java.

Listed below are the topics covered in this article:

What is Package in Java?

Java package is a mechanism of grouping similar type of classes, interfaces, and sub-classes collectively based on functionality. When software is written in the Java programming language, it can be composed of hundreds or even thousands of individual classes. It makes sense to keep things organized by placing related classes and interfaces into packages.

Using packages while coding offers a lot of advantages like:

- **Re-usability:** The classes contained in the packages of another program can be easily reused
- **Name Conflicts:** Packages help us to uniquely identify a class, for example, we can have *company.sales.Employee* and *company.marketing.Employee* classes
- **Controlled Access:** Offers access protection such as protected classes, default classes and private class
- Data Encapsulation**:** They provide a way to hide classes, preventing other programs from accessing classes that are meant for internal use only
- **Maintainance:** With packages, you can organize your project better and easily locate related classes

It's a good practice to use packages while coding in Java. As a programmer, you can easily figure out the classes, interfaces, enumerations, and annotations that are related. We have two types of packages in java.

Types of Packages in Java

Based on whether the package is defined by the user or not, packages are divided into two categories:

1. Built-in Packages
2. User Defined Packages

Built-in Packages

Built-in packages or predefined packages are those that come along as a part of JDK (Java Development Kit) to simplify the task of Java programmer. They consist of a huge number of predefined classes and interfaces that are a part of Java API's. Some of the commonly used built-in packages are java.lang, java.io, java.util, java.applet, etc. Here's a simple program using a built-in package.

```
1   package Edureka;

2   import java.util.ArrayList;

3    class BuiltInPackage

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

```
5     ArrayList<Integer> myList = new ArrayList<>(3);
6          myList.add(3);
7          myList.add(2);
8          myList.add(1);
9          System.out.println("The elements of list are: " + myList);
10    }
11  }
```

Output:

```
1     The elements of list are: [3, 2, 1]
```

The ArrayList class belongs to java.util package. To use it, we have to import the package using the import statement. The first line of the code *import java.util.ArrayList* imports the java.util package and uses ArrayList class which is present in the sub package util.

User Defined Packages

User-defined packages are those which are developed by users in order to group related classes, interfaces and sub packages. With the help of an example program, let's see how to create packages, compile Java programs inside the packages and execute them.

Creating a Package in Java

Creating a package in Java is a very easy task. Choose a name for the package and include a *package* command as the first statement in the Java source file. The java source file can contain the classes, interfaces, enumerations, and annotation types that you want to include in the package. For example, the following statement creates a package named ***MyPackage.***

```
1       package MyPackage;
```

The package statement simply specifies to which package the classes defined belongs to..

Note: *If you omit the package statement, the class names are put into the default package, which has no name. Though the default package is fine for short programs, it is inadequate for real applications.*

Including a Class in Java Package

To create a class inside a package, you should declare the package name as the first statement of your program. Then include the class as part of the package. But, remember that, a class can have only one package declaration. Here's a simple program to understand the concept.

```
1   package MyPackage;
2    public class Compare {
3     int num1, num2;
4      Compare(int n, int m) {
```

```
5       num1 = n;
6       num2 = m;
7   }
8   public void getmax(){
9      if ( num1 > num2 ) {
10        System.out.println("Maximum value of two numbers is " + num1);
11    }
12     else {
13        System.out.println("Maximum value of two numbers is " + num2);
14     }
15  }
16
17
18  public static void main(String args[]) {
19        Compare current[] = new Compare[3];
20
21        current[1] = new Compare(5, 10);
22        current[2] = new Compare(123, 120);
23
24          for(int i=1; i < 3 ; i++)
25              {
26              current[i].getmax();
27              }
28       }
29  }
30
31
```

Output:

```
1    Maximum value of two numbers is 10

2    Maximum value of two numbers is 123
```

As you can see, I have declared a package named MyPackage and created a class Compare inside that package. Java uses file system directories to store packages. So, this program would be saved in a file as *Compare.java* and will be stored in the directory named MyPackage. When the file gets compiled, Java will create a **.class** file and store it in the same directory. Remember that name of the package must be same as the directory under which this file is saved.

You might be wondering how to use this Compare class from a class in another package?

Creating a class inside package while importing another package

Well, it's quite simple. You just need to import it. Once it is imported, you can access it by its name. Here's a sample program demonstrating the concept.

```
1    package Edureka;

2    import MyPackage.Compare;

3

4    public class Demo{

5        public static void main(String args[]) {

6            int n=10, m=10;

7            Compare current = new Compare(n, m);

8            if(n != m) {

9                current.getmax();

10           }

11           else {

12               System.out.println("Both the values are same");

13           }

14   }

15   }
```

Output:

```
1        Both the values are same
```

I have first declared the package *Edureka*, then imported the class *Compare* from the package MyPackage. So, the order when we are creating a class inside a package while importing another package is,

- Package Declaration
- Package Import

Well, if you do not want to use the import statement, there is another alternative to access a class file of the package from another package. You can just use fully qualified name while importing a class.

Using fully qualified name while importing a class

Here's an example to understand the concept. I am going to use the same package that I have declared earlier in the blog, *MyPackage*.

```
1   package Edureka;
2   public class Demo{
3       public static void main(String args[]) {
4           int n=10, m=11;
5           //Using fully qualified name instead of import
6           MyPackage.Compare current = new MyPackage.Compare(n, m);
7           if(n != m) {
8               current.getmax();
9           }
10          else {
11              System.out.println("Both the values are same");
12          }
13  }
14  }
```

Output:

```
1   Maximum value of two numbers is 11
```

In the Demo class, instead of importing the package, I have used the fully qualified name such as *MyPackage.Compare* to create the object of it. Since we are talking about importing packages, you might as well check out the concept of static import in Java.

Static Import in Java

Static import feature was introduced in Java from version 5. It facilitates the Java programmer to access any static member of a class directly without using the fully qualified name.

```
1   package MyPackage;
2   import static java.lang.Math.*; //static import
```

```
3    import static java.lang.System.*;// static import
4    public class StaticImportDemo {
5        public static void main(String args[]) {
6            double val = 64.0;
7            double sqroot = sqrt(val); // Access sqrt() method directly
8            out.println("Sq. root of " + val + " is " + sqroot);
9            //We don't need to use 'System.out
10       }
11   }
```

Output:

```
1        Sq. root of 64.0 is 8.0
```

Though using static import involves less coding, overusing it might make program unreadable and unmaintainable. Now let's move on to the next topic, access control in packages.

Access Protection in Java Packages

You might be aware of various aspects of Java's access control mechanism and its access specifiers. Packages in Java add another dimension to access control. Both classes and packages are a means of data encapsulation. While packages act as containers for classes and other subordinate packages, classes act as containers for data and code. Because of this interplay between packages and classes, Java packages addresses four categories of visibility for class members:

- Sub-classes in the same package
- Non-subclasses in the same package
- Sub-classes in different packages
- Classes that are neither in the same package nor sub-classes

The table below gives a real picture of which type access is possible and which is not when using packages in Java:

	Private	***No Modifier***	***Protected***	***Public***
Same Class	Yes	Yes	Yes	Yes
Same Package Subclasses	No	Yes	Yes	Yes
Same Package Non-Subclasses	No	Yes	Yes	Yes
Different Packages Subclasses	No	No	Yes	Yes

Different Packages Non-Subclasses	No	No	No	Yes

We can simplify the data in the above table as follows:

1. Anything declared public can be accessed from anywhere
2. Anything declared private can be seen only within that class
3. If access specifier is not mentioned, an element is visible to subclasses as well as to other classes in the same package
4. Lastly, anything declared protected element can be seen outside your current package, but only to classes that subclass your class directly

This way, Java packages provide access control to the classes. Well, this wraps up the concept of packages in Java. Here are some points that you should keep in mind when using packages in Java.

Points to Remember

- Every class is part of some package. If you omit the package statement, the class names are put into the default package
- A class can have only one package statement but it can have more than one import package statements
- The name of the package must be the same as the directory under which the file is saved
- When importing another package, package declaration must be the first statement, followed by package import

Well, this brings us to the end of this 'Packages in Java' article. We learned what a package is and why we should use them. There is no doubt that Java package is one of the most important parts for efficient java programmers. It not only upgrades the programmer's coding style but also reduces a lot of additional work.

If you found this article on "Packages in Java", check out the Java Training by Edureka, a trusted online learning company with a network of more than 250,000 satisfied learners spread across the globe. We are here to help you with every step on your journey, for becoming a besides this java interview questions, we come up with a curriculum which is designed for students and professionals who want to be a Java Developer.

How to access Java package from another package

You can understand it using an example where a Boss class is defined in payroll package.

```
package payroll;
public class Boss {
  public void payEmployee(Employee e) {
    e.mailCheck();
  }
}
```

if the Employee class is not in the payroll package? The Boss class must then use one of the following techniques for referring to a class in a different package.

- The fully qualified name of the class can be used. For example −

```
payroll.Employee
```

- The package can be imported using the import keyword and the wild card (*). For example −

```
import payroll.*;
```

- The class itself can be imported using the import keyword. For example −

```
import payroll.Employee;
```

A class file can contain any number of import statements. The import statements must appear after the package statement and before the class declaration.

Access Specifiers In Java

Do you like this Article?
Definition :

- Java Access Specifiers (also known as Visibility Specifiers) regulate access to classes, fields and methods in Java.These Specifiers determine whether a field or method in a class, can be used or invoked by another method in another class or sub-class. Access Specifiers can be used to restrict access. Access Specifiers are an integral part of object-oriented programming.

Types Of Access Specifiers :

In java we have four Access Specifiers and they are listed below.

1. public
2. private
3. protected
4. default(no specifier)

We look at these Access Specifiers in more detail.

Access Modifiers	Default	private	protected	public
Accessible inside the class	yes	yes	yes	yes
Accessible within the subclass inside the same package	yes	no	yes	yes
Accessible outside the package	no	no	no	yes
Accessible within the subclass outside the package	no	no	yes	yes

public specifiers :

Public Specifiers achieves the highest level of accessibility. Classes, methods, and fields declared as public can be accessed from any class in the Java program, whether these classes are in the same package or in another package.

Example :

```
public class Demo {  // public class
public x, y, size;   // public instance variables
}
```

private specifiers :

Private Specifiers achieves the lowest level of accessibility.private methods and fields can only be accessed within the same class to which the methods and fields belong. private methods and fields are not visible within subclasses and are not inherited by subclasses. So, the private access specifier is opposite to the public access specifier. Using Private Specifier we can achieve encapsulation and hide data from the outside world.

Example :

```
public class Demo { // public class

private double x, y;   // private (encapsulated) instance variables

public set(int x, int y) {  // setting values of private fields
this.x = x;
this.y = y;
}

public get() {  // setting values of private fields
return Point(x, y);
}
}
```

protected specifiers :

Methods and fields declared as protected can only be accessed by the subclasses in other package or any class within the package of the protected members' class. The protected access specifier cannot be applied to class and interfaces.

default(no specifier):

When you don't set access specifier for the element, it will follow the default accessibility level. There is no default specifier keyword. Classes, variables, and methods can be default accessed.Using default specifier we can access class, method, or field which belongs to same package,but not from outside this package.