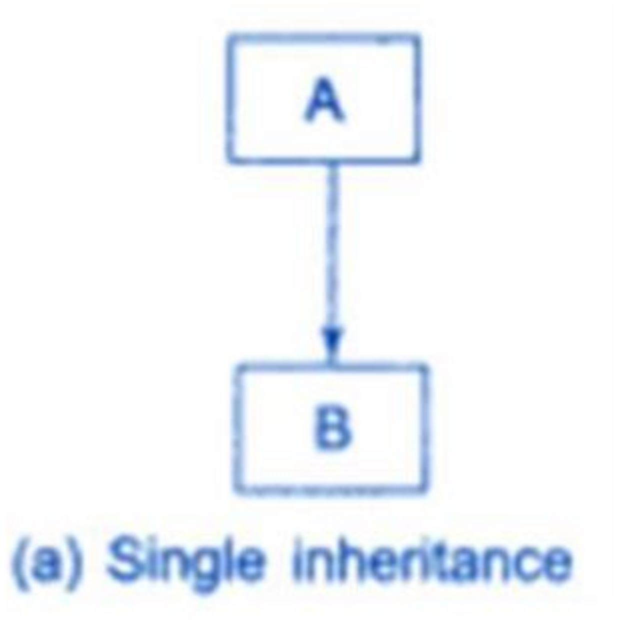
JAVA UNIT – 2

Types of Inheritance

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance

Single Inheritance

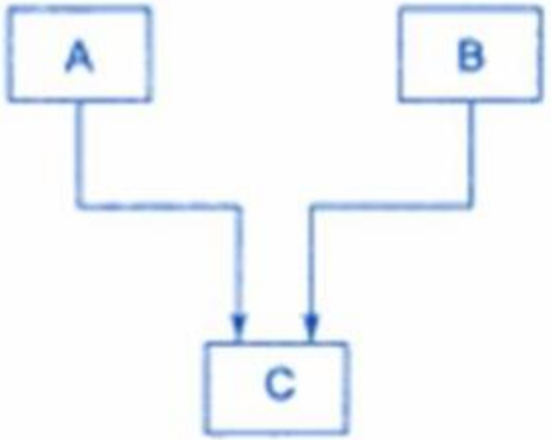
- A derived class with only one base class is called single inheritance



(a) Single inheritance

Multiple Inheritance

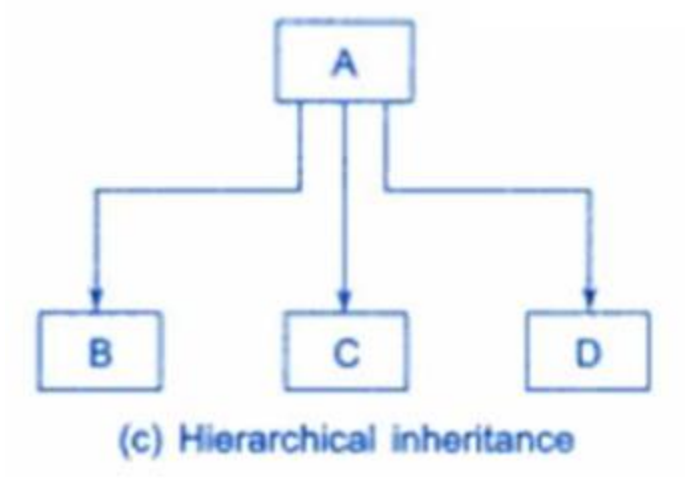
- A derived class with multiple base class is called multiple inheritance.



(b) Multiple inheritance

Hierachical Inheritance

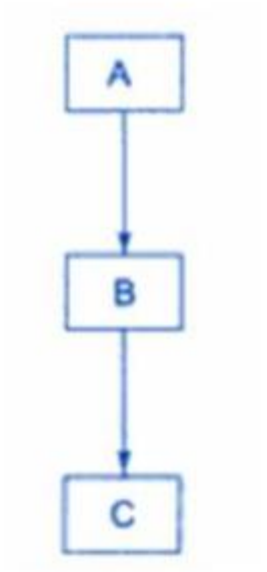
- A single class may be inherited by more than one class is called Hierarchical inheritance.



(c) Hierarchical inheritance

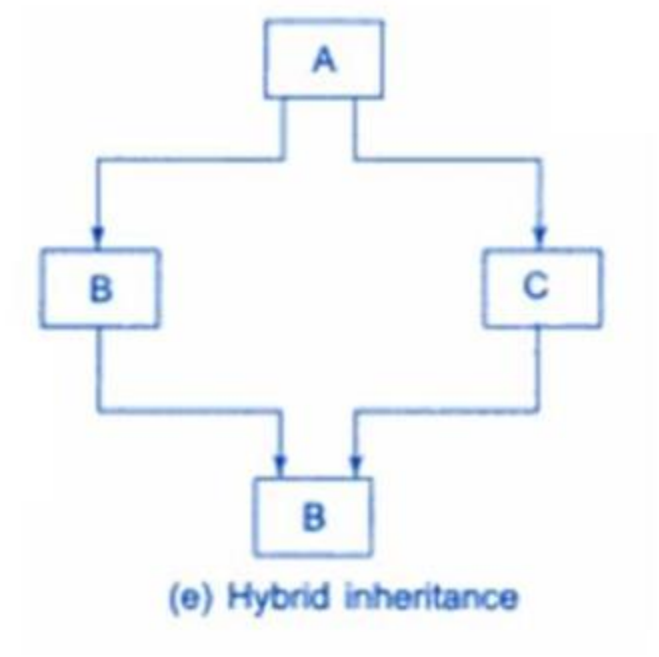
Multilevel Inheritance

- The mechanism of deriving a class from another "derived class" is known as multilevel inheritance



Hybrid Inheritance

- The combination of various types of inheritance is known as hybrid inheritance



(e) Hybrid inheritance

Defining a Sub class

- The general form is given below

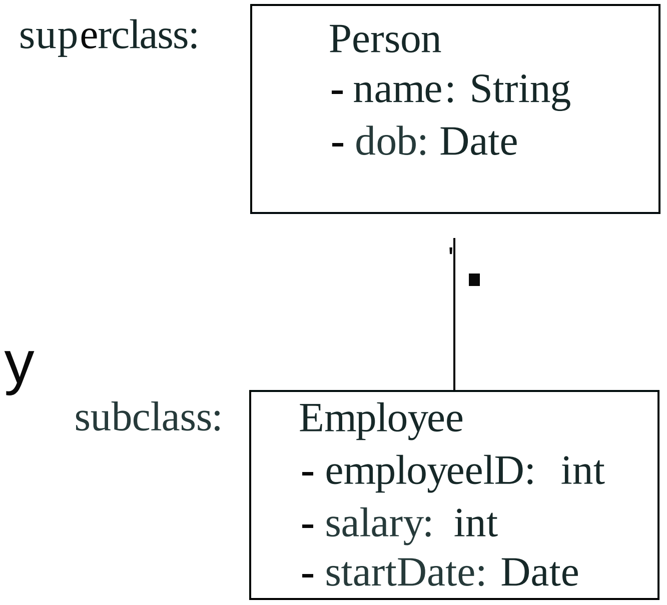
```
class sub-class-name extends super class name
{
        //members of derived class
}
```

The keyword extends signifies that the properties of the Super class name are extended to the sub class name.

- Inheritance is a fundamental Object Oriented concept

- A class can be defined as a "subclass" of another class.
  - The subclass inherits all data attributes of its super class
  - The subclass inherits all methods of its super class
  - The subclass inherits all associations of its super class

- The subclass can:
  - Add new functionality
  - Use inherited functionality
  - Override inherited functionality



```
class base

{

    int temp;

    void display()

    {

    System.out.println("I m from base");

    }

}
class derived extends base

{

    void display1()


    System.out.println("I m from drv");
    }
```

```
class test
{
    public static void
    main (String argc[])
   { derived d=new
     derived();

     d.display();//superclass
     d.displayl();//subclass
    }
 }
```

Using super to use the constructor of a super class

- A sub class does not inherit constructor from the super class

- So we need to define the constructor for derived class

- We can use super keyword to invoke the constructor of super class

```
class base
{
    int temp;
    base(int b)
    {
        temp=b;
    }
}
class derived extends base
{
    int temp1;
    derived(int d)
    {
        super(d);
        temp1=d;
    }
}
```

```
class test
{
    public static void
    main (String args[])
    {
        derived d=new
        derived(5);
    }
}
```

- The first statement in any constructor is either this or super.
- When we do not specify either this or super as the first statement in a derived class constructor, it is assumed to be a super with no parameters.
- For example,

```
derived(int d)
{
        temp1=d;

 }
```

```
derived(int d)
{

        super();
        temp1=d;
}
```

- **this** keyword can be used to invoke the constructor of the same class

```
derived(int temp1,temp2,int d)
{
        super();
        temp1=d
}
```

```
  derived(int d)
{
        this ( 1,1,1);
}
```

derived d=new derived(3)

The first statement in the constructor with one parameter is this, which invokes a constructor of the same class

It would in turn invoke the constructor of the super class first

Method overriding and use of super

- Method overriding is a mechanism where a sub class defines a method with the same name and same method signature, available in a super class

- In other words When both the classes(super and sub) having a method witll same name and same signature, then that method is known as overridden method

```
class base
{
   void display()
   {
       System.out.println("i
          am from base");
   }
}
class derived extends base
{
   void display()
   {
         System.out.println("I
         am from derived");
   }
}
```

```
class test
{
     public static void
     main (String args[])
    {
      derived d=new derived();
       d.display();
      base b=new base();
         b.display();
         b=new derived();
         b.display();
    }
}
```

Using super keyword to invoke a method of super class

- Super keyword can be used to invoke a method of the super class

```
class base
{
   void display()
   {
     System.out.println("I
          from base");
   }
}
class derived extends base
{
  void display()
  {
     System.out.println("I
      am from derived”);
      super.display();
  }
 }
```

```
class test
{
    public static void
    main (String args[])
   {
     derived d=new derived();
       d.display();
   }
 }
```

Variable shadowing and the use of super

- In inheritance relationship the instance members from super class inherited in sub class

- Variable shadowing is a term used for situations, when child and parent classes have variable with exactly same name and data type.

Method and Variable Bindinding

```
class base
{
   int t=4;
   void display()
   {
     System.out.println("base");
   }
}
class derived extends  base
{
 int t=5; // shadow variable
 void display()
  {
     System out.println("deri");
   }
}
```

```
class test
{
     public static void
     main (String args[])
    {
      derived d=new derived();
       d.display();
      base b=new base();
       b.display();
       b=new derived();
       b.display();
       System.out.println(b.t);
    }
}
```

Using final with variables

- The keyword final is used with a variable to declare a constant ,
- i.e. if a variable is declared to be final then that variable can not be modified.
- All final variables must be initialized , before they can be used
- For Example,

```
final int size=101;
```

”

Using final with variables

- Final modifier may be used with static ,instance , parameter and local variables

- The final modifier may be used for variables of any type i.e. primitive or reference types

- In case of variable that is reference type, the reference can not be changed

- Once a reference final variable is initialized , and refers to an instance, than that reference variable can not refer to any other instance.

Using final with variables

```
//this code will work
    rectangle r=new rectangle();
    r.display();
    r=new access();

//this code will throw an error
    final rectangle r=new rectangle();
    r.display();
    r=new access();//this statement will give
    error.
```

Using final with variables

- All methods can be overridden by default in sub classes.

- If we wish to prevent the sub classes from overriding the members of the super class, we can declare them as final using the keyword final as a modifier.

- For Example :

    Final void showstatus();

Using final with variables and methods

```
class rectangle
{
    final void display()
    {
        System.out.println( "Hello world");
    }
}
class access extends rectangle
{
    void display() //this will cause an error
    {

    }
```

Using final with classes

- Sometimes we may like to prevent a class being further subclasses for security reasons.
- A class that can not be sub classed is called a final class
- This can be achieved using a final keyword as

  final class Aclass { .....}

  class Bclass extends Aclass { .... } (it not work)

Any attempt to inherit these classes will cause an error and the compiler will not allow it.

Abstract Classes and Abstract Methods

- We have seen that by making a method final we ensure that the method is not redefined in a sub classes.
- That is the method can not be sub classed.
- Java allows us to do exactly opposite to this.
- That is , we can indicate that a method must always be redefined in a sub class , thus making overriding compulsory.

Abstract Classes and Abstract Methods

- This is done using the modifier keyword abstract in the method definition

  For example,

```
    abstract class vehicle
  {
      // an abstract class may  have abstract method

      abstract void start();
      // a bstract methods do not have an
     // implementation
```

- While using abstract classes, we must satisfy the following condition

l 1) we can not use abstract classes to instantiate objects directly

For Example,

```
vehicle v=new vehicle();
//error because vehicle is an abstract class
```

'(

2) The abstract method of an abstract class must be defined in its sub elass

3) we cant declare abstract constructors or abstract static methods(abstract keyword can be used)

- Even though we do not create instances of an abstract class, an abstract class can always have a constructor.

- The constructor of an abstract class will be used from the constructor of its sub classes.

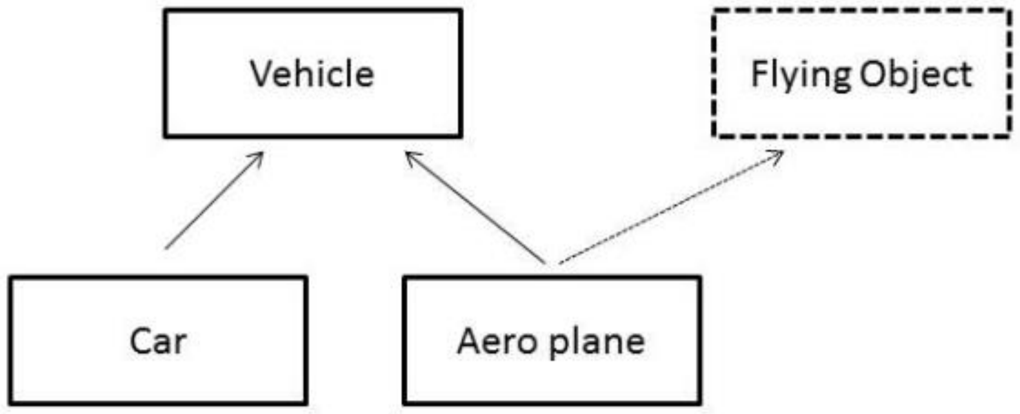
Single inheritance of classes

- In java, we do not have multiple inheritance of a classes.

- A class can not have More than one direct super class.

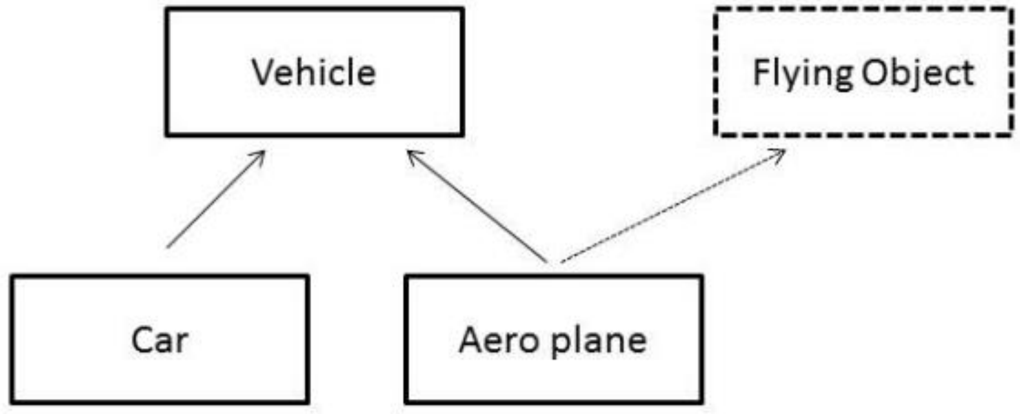
,.

- When we define the class, we can specify only one class name in the extends clause of the class definition.



- In the above example, we have a Aero Plane, which has two super types. One , vehicle and another Flying Object

- Now this flying object can be defined as abstract as different types of flying Object will use different things for flying.



- Flying object will be a type, which has no instance variables and has only abstract method (assume fly() method as an abstract)
- When we have such data types that have no instance variables and only abstract methods, then these type can be defined as interfaces.

- Java does not supports multiple inheritance
- That is classes in Java cant have more than one super classes.

For Example,

```
class A extends B extends c

{
      .....

}
```

Is not permitted in java

- However there may be a situation where multiple inheritance required to inherit methods and properties from several classes

- Java provides and alternative approach known as interfaces to support the concept of multiple inheritance

- Although a java class cant be a subclass of more than one super class, it can implement more than one interface.

Defining Interfaces

- An interface is basically a kind of class.
- Like classes , interfaces contain methods and variables but with a major difference

- The difference is that interfaces do not specify any code to implement these methods and instance variables contain only constants.

- Therefore , it is the responsibility of the class that implements an interface to define the code for implementation of these methods

The general form

```
interface inteface_name
{
        variable declaration;
        methods declaration;
 }
```

For Example,

```
interface Item
{
        static final int code = 1001;
        void display();
}
```

Extending Interfaces

- Like classes, interfaces can also be extended.
- That is an interface can be sub interfaced from other interfaces.

```
interface name2 extends name1
 {
        body of name2
 }
```

- The new sub interface will inherit all the members of the super interface in a manner similar to a class.

For Example,

```
Interface ItemConstants
{
        int code=1001;
        string name="Fan";
}
Interface Item extends ItemConstants
{
        void display();
}
```

```
interface  ItemConstants
{
        int code=1001;
        string name="Fan";
}
interface   ItemMethods
{
        void display();
}
Interface Item extends ItemConstants,ItemMethods
{
        ………………………….
}
```

we can combine several interfaces together into a single interface.

Implementing Interfaces

- Interfaces are used as "superclasses" whose properties are inherited by classes
- It is therefore necessary to create a class tflat inherits the given interface.

```
class classname implements interface_name
{
    body of classname
}
```

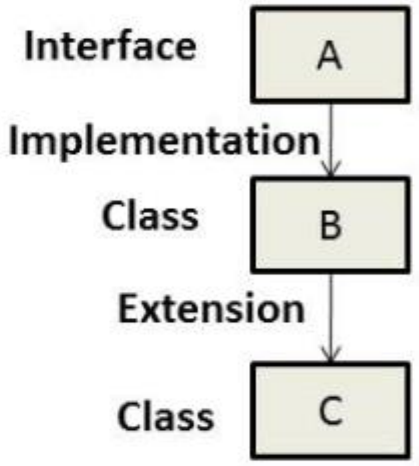
```
class classname extends super_class
implements interface1,inteface2
{
        body of classname
}
```

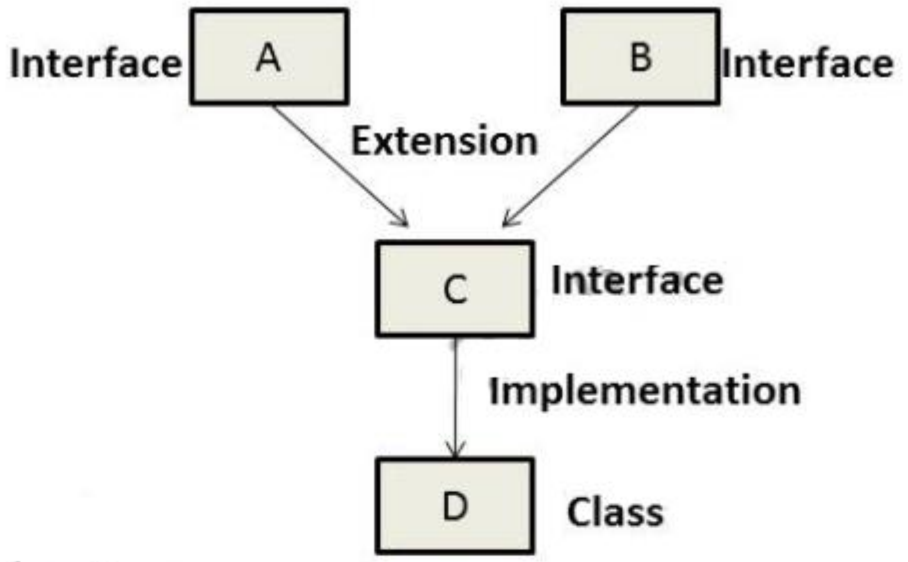
For example,

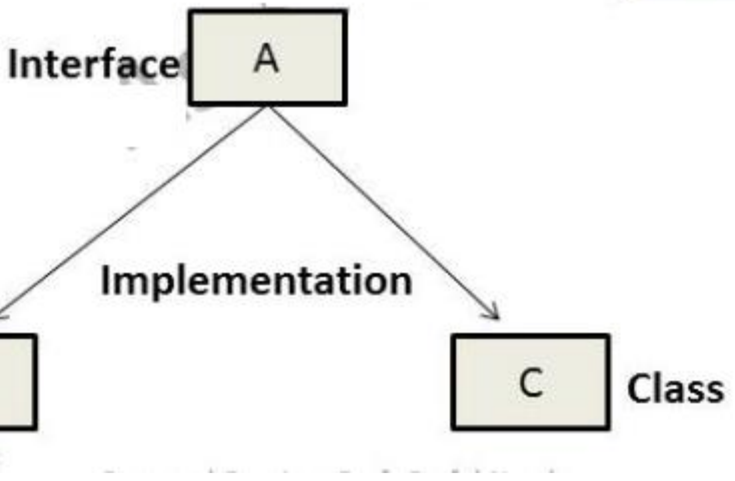
```
class child_class extends base_class implements
   item_inteface,product_interface
{

}
```

- **The implementation of interfaces can take various forms as in figures**







```
interface Area
{
        final static float pi=3.14f;
        float coumpute(float x,float y);
}
class  Rectangle implements Area
 {
        public float compute(float x,float y)
        {
                return(x*y);
        }
}
```

```
class   Circle implements Area
{
          public float compute(float x,float y)
          {
                    return(pi*x*y);
          }
}
class  InterfaceTest
{         public static void main(String args[])
          {
                    float result1,result2;
                    rectangle r=new rectangle();
                    circle c=new circle();
                    result1 =r.compute (10,20);
                    result 2=c.compute(10,2);
          }
```

The Object class

- All Java objects derive from the Object base class. Even though you don't explicitly include the "extend Object" in your class definition.
- For example,

```
public class Test
{
    ...
}
```

is equivalent to

```
public class Test extends Object
 {
    •••
 }
```

- This means that all Java objects are of the Object type.
- This allows for polymorphic referencing throughout.
- All reference types are assignable to a variable of object type, including arrays and interfaces
- For Example,

```
Object obj;
obj=new int[10]; // it is valid
```

Access Specifiers

- Java programming language provides access specifiers to allow or restrict the use of various members from other class definitions and interface definitions

- In java we have total four access specifier

  1)public

  2)private

  3)protected

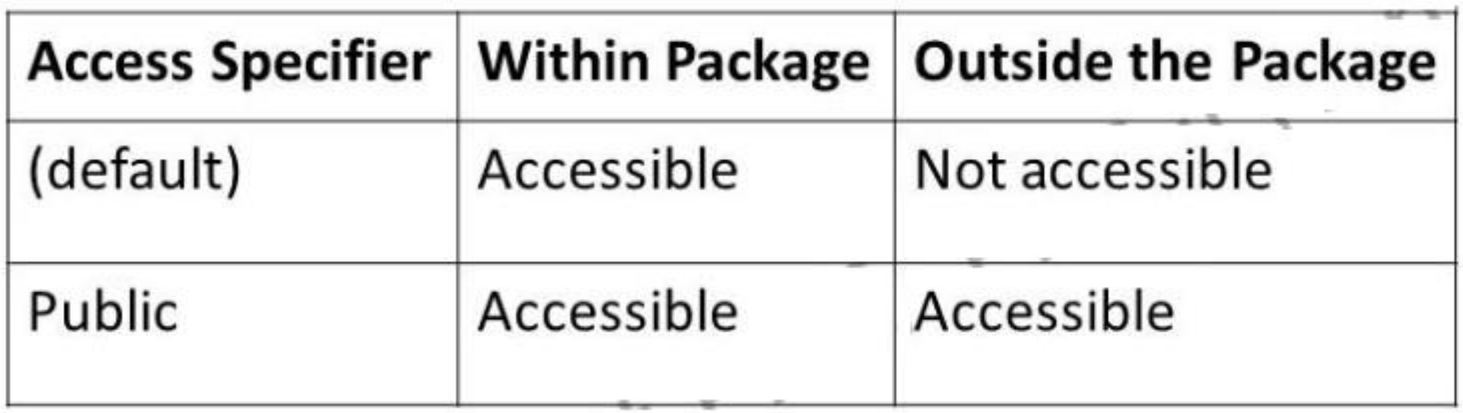
  4)default

Access Specifiers for member of a package

- For the classes and interfaces that are a part of a package, there are only two access specifiers
- They are default and public
- When any member of a package has a default access specifier ,then such a member is known and useable within its package.

- When any member of a package has a public access specifer, then such a member is known everywhere

Access Specifiers for member of a package



Access Specifier	Within Package	Outside the Package
(default)	Accessible	Not accessible
Public	Accessible	Accessible

Access specifier for a members of class

- For the members of class there are four access specifiers available

1) Private Access specifier

- When any member of a class is declared to be private then that private member is accessible only within class

private members are not inherited by a sub class

Access specifier for a members of class

2) Default Access Specifer

- When any member of a class is declared with a default access specifier, then that member is accessible from anywhere within the package to which the class belongs.

3) Public Access specifer

- When any member of a public class is declared to be public ,then that members has no access restrictions.

Access specifier for a members of class

4) Protected Access Specifier

- protected methods and fields can only be accessed within the same class to which the methods and fields belong, within its subclasses, and within classes of the same package, but not from anywhere else.

- You use the protected access level when it is appropriate for a class's subclasses to have access to the method or field, but not for unrelated classes.

Access specifier for a members of class

Access Specifier	Within class	Within Package	Sub classes outside the package	Non-sub classes outside the package
Private	A	NA	NA	NA
(default)	A	A	NA	NA
protected	A	A	Not accessible but inherited	NA
Public	A	A	A	A

Access Specifier public Example

```
class ParentClass
{   public void test()
    {
        System.out.println("This is my exampie");
    }
}
public class AccessSpecifier
{
    public static void main(String[] args)
    {
        parentClass accessSpecifier = new ParentClass();
        accessSpecifier.test();
    }
}
```

Access Specifier private Example

```
class ParentClass
{
    private int a = 10;
    public void showValue()
    {
     System.out.println("a value " +
    a);
    }
     private void test()
    {
         System.out.println("This is
         my example");
     }
   }
```

```
public class AccessSpecifier
{
  public static void
  main(String[] args)
   {
          ParentClass p= new
          ParentClass();
          p.test();
          p.a = 5;   (not work)

          p.showValue();
          //works properly
     }
}
```

Access Specifier protected Example

```
class ParentClass
{
    protected int a = 10;
    protected void test()
    {
        System.out.println("This is
        my example");
    }
}
class childclass  extends
    ParentClass
{
}
```

```
public class AccessSpecifier
{
  public static void
  main(String[] args)
  {
        childclass c = new
        childclass();
        c.test();
        c.a =  5;
        //works properly
  }
}
```

Creating packages

- What if we need to use classes from other programs without physically copying them into the program under development?

- This can be accomplished in java by using what is known as packages, a concept similar to "class libraries" in other language.

- It is another way of achieving the reusability in java

What is Package?

- Packages are java's way of grouping a variety of classes and interfces together.

- The grouping is done according to the functionality.

- In fact, packages act as "containers" for classes and interfaces.

Benefits of using Package

1) the classes contained in the packages of other program can be easily used.

2) In packages, classes can be unique compared with classes in other packages. That is , two classes in two different packages can have the same name.

They may be referred by their fully qualified name , comprising the package name and the class name

Benefits of using Package

3) Packages provide a way to "hide" classes thus preventing other programs of packages from accessing classes that are meant for internal use only

4) Packages also provide a way for separating "design" from "coding"

First we can design classes and decide their relationships, and then we can implement the java code needed for the methods.

packages

- To create a package , we must first declare the name of the package using the package keyword followed by a package name
- For example,

```
package firstPackage;          // pkg.declaration

public class FirstClass
{
        ......
}
```

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

Compilation of program

```
C:\pro\mypackage\javac TestRectangle.java
C:\pro\java TestRectangle
```

Packages

- We have repeatedly stated that one of the main features of OOP is its ability to reuse the code already created.

- One way of achieving this is by extending the classes and implementing the interfaces we had created .

- This is limited to reusing the classes within a program.

Overall process of defining a Package

- To create a package , simply include a package command as the first statement in java source file

- Any classes declared within that file will belong to the specified package.

- The package statement defines a name space in which classes are stored.

- If you omit the package statement, the class names are put into the default package, which has no name.

Hierarchy of package

- You can create the hierarchy of package.
- To do so, simply separate each package name from the one above it by use of a period.
- The general form of multileveled package statement

```
package pkg1[.pkg2[.pkg3]];
```

For example,

```
package mypackage.studentinfo.bca;
```

How to use package?

There are two ways in order to use the public classes

1. Declare the fully-qualified class name.

   For example,

```
public class Example
{
    public static void main(String args[])
    {
          mypackage.TestRectangle   temp=new
           mypackage.TestRectangle();
    }
```

How to use package?

2) Use an "import" keyword:

```
import mypackage.*;
public class Example
{
        public static void main(String argc[])
        {

              TestRectangle temp =new TestRectangle();

        }
 }
```

More on import statement

- To use a package's classes inside a Java source file, it is convenient to import the classes from the package with an import declaration.
- The following declaration

import java.awt.event.*;

- imports all classes from the java.awt.event package,

While the next declaration

import java .awt.event.ActionEvent;

- import only the ActionEvent class from the package.

Use of static imports

- Static import is another language feature introduced with the J2SE 5.0 release.
- This feature eliminates the need of qualifying a static member with the class name.
- The static import declaration is similar to that of import.
- we can use the static import statement to import static members from classes and use them without qualifying the class name.

For Example,

```
package employee.emp_details;
public interface sal_increment
{
    public static final double manager= 0.5;
    public static final double clerk= 0.25;
}
```

```
import static employee.emp_details.sal_increment;
class salary_hike
{
        public static void main(String args[])
        {
                double m_sal=manager*25000;
        }
 }
```

Static member without interface qualifier name

Thus, we can use static member in the code without qualifying the class name or interface name.

Using the JAVA APIs

- Java API is actually a huge collection of library routines that performs basic programming tasks. such as looping, displaying GUI form etc.

The Java comprises three components:

- Java Language
- JVM or Java Virtual Machine and
- The Java API (Application programming interface)

Using the JAVA APIs

- Type of Java API

There are three types of API available in Java Technology.

1) Official Java Core API

The official core API is part of JDK download. The three editions of the Java programming language are Java SE, Java ME and Java EE.

2) Optional Java API

The optional Java API can be downloaded separately.

3) Unofficial APIs

These API's are developed by third parties and can download from the owner website.

Name	Packages that contain the API
Abstract WindowToolkit	java.awt
Swing	javax.swing
Accessibility	javax.accessibility
Drag n Drop	java.awt.datatransfer java.awt.dnd
Image I/O	javax.imageio Javax.imageio.*
Sound	javax.sound.midi javax.sound.midi.spi
java Database Connectivity	java.sql javax.sql
java Cryptography Extension	javax.crypto javax.crypto.interfaces javax.crypto.spec
java Authentication and Authorization Service	javax.security.auth
java Secure Socket Extension	javax.net javax.net.ssl java.security.cert

Commonly Used Classes from the java.lang package

- There are various packages available as part of the java API
- One of the basic packages is the java. lang paekage
- This package contains the core classes and interfaces used in the java language.
- This includes classes and interfaces like System, String, StingBuffer,StringBuilder,Comparable,Math and Wrapper classes.

Comparable Interface

- The relational operator < > <= >=works only for numeric data types i.e. the comparison is not available for reference data types

- Comparable interface can be used to compare object's greater than or less than relations

- The comparable interface is always used to indicate that instances of a particular class have a natural ordering.

Comparable Interface

- The logic for making the comparison depends upon the class
- Each class which is being defined as comparable has its own logic for comparison.
- This can be specified by implementing the comapreTo method from he Comparable interface
- It has been defined as follows

```
public interface Comaprable
{
    public int compareTo(Object o);
}
```

How it is used ?

java. lang.Comparable: int compareTo(Object o1)

This method compares this object with o1 object.

Returned int value has the following meaning

1) positive - this object is greater than o1
2) zero - this object equals to o1
3 negative - this object is less than o1

```
public class Employee implements Comparable
{
    private int empId;
    public Employee(int i)
    {
        empId=i;
    }
    public int compareTo(Object o)
    {
        return this.empId - ((Employee)o).empId;
    }
    public static void main(String [] args)
    {
        Employee e1=new Employee(2);
        Employee e2=new Employee(2);
        System.out.println(e1.compareTo(e2));
    }
```

The comparator Interface

- The comparator is an interface in the java.util package.
- In case of the comparable interface, we have the compareTo() method, which takes qnly one parameter.
- The comparator interface has only one method called compare() which takes two parameters.

```
public interface Comparator
{

    public int compare(Object o1,Object o2);
}
```

The comparator Interface

- An instance of comparator interface is used for specifying the logic for ordering objects.
- we may have a requirements of ordering objects in different manners
- eg. For the instance of employee class We may like to order the instances of Employee by empid or by employee salary.
- this is where the comparator interface can be useful.

How is it used?

- **java.util.Comparator : int compare(Object o1 , Object o2)**

  This method compares o l and o2 objects.

- Returned int value has the following meanings.

  1) positive – o1 is greater than o2
  2) zero - o1 equals to o2
  3) Negative – o1 is less than o1

Comparator Example

```
class Employee
{
    private int age;
    private String name;
    public void setAge(int
    age)
    {
        this.age=age;
    }
    public int getAge()
    {
        return this.age;
    }
```

```
public void setName(String
name)
    {
        this.name=name;
    }

    public String getName()
    {
        return this.name;
    }
    }
```

```
class AgeComparator implements Comparator
{
  public int compare( Object emp1, Object emp2)
  {
    int emp1Age = ((Employee)emp1).getAge();
    int emp2Age = ((Employee) emp2).getAge();
    if(emp1Age > emp2Age)
      return 1;
    else if(emp1Age < emp2Age)
      return -1;
    else
      return 0;
    }
 }
```

```
class NameComparator implements Comparator
{
    public int compare( Object emp1, Object emp2)
    {
            String emp1Name = ((Employee)emp1).getName();
            String emp2Name= ((Employee) emp2).getName();
            return emp1Name.compareTo(emp2Name);
    }
}
```

```
public class JavaComparatorExample
{   public static void main(String args[])
    {     Employee employee[] = new Employee[2];
          employee[0] = new Employee();  employee[0]
          .setAge(40);
          employee[0].setName("ABC");
          employee[1] = new Employee();
          employee[1].setAge(20);
          employee[1].setName("XYZ");
          for(i  t i=0; i< employee.length; i++){
          System.out.println("Employee " + (i + 1) + " name :: " +
    employee[i].getName() + ", Age :: " + employee[i].getAge());
          }
          }
```

String Class

- String is most commonly used class

- In fact every data type can be converted to String.

- String is a sequence of Unicode characters.

- String is a final Class, whose instances are immutable

- That means we can not create sub classes from the String class and once any instances created, there is no-way that we could change the characters in the String instances.

String Class

String constants

- String constants can be used in any class by enclosing a sequence of characters within double quotes.

For Example,

- String s1="hello";
- String s2= new String(s1);
- String s3= "Hello" +“word” //Concatenation
- String s4="hello";

- If two or more Strings have the same set of characters in the same sequence then they share the same reference in memory.

String Class

- **Arrays Of string**
- You can create arrays of strings as given below

```
String [] names=new String[5];
```

You can initialize the array as given below

```
String[] colors={"red“,”orange”,”yellow”};
```

String Class

- **The string constructors**

1) String()

2) String(char charArray[])

   For example,

```
char name[]={'j','a','v','a'};
String sname=new String(name);
```

3) String(char cArray[],int startIndex,int no_ofcharacters)

   For example,

```
char name[]={'j','a','v','a'};
String sname=new String(name,2,2);
```

String Class

The string constructors

4)String(String object)

S)String(byte byteArray[])

For example,

```
byte byteArray[]={65,66,67},

String sname=new String(byteArray);
```

6)String(byte byteArray[],int startIndex,int no_of_char)

StringBuilder and StringBuffer class

- StringBuilder objects are like String objects, except that they can be modified.

- Internally, these objects are treated like variable length arrays that contain a sequence of characters.

- At any point, the length and content of the sequence can be changed through method invocations.

StringBuilder and StringBuffer class

- **java .lang.StringBuilder**
- **SringBuilder** class is introduced in ***Java 5.0*** version. This class is replacement for the existing ***StringBuffer*** class.

- If you are using ***StringBuilder* , *no*** guarantee of ***synchronization* .**

,.

- This is the only difference with ***StringBuffer*** class.

- It is recommended that this class be used in preference to ***StringBuffer*** as it will be faster.

For Example,

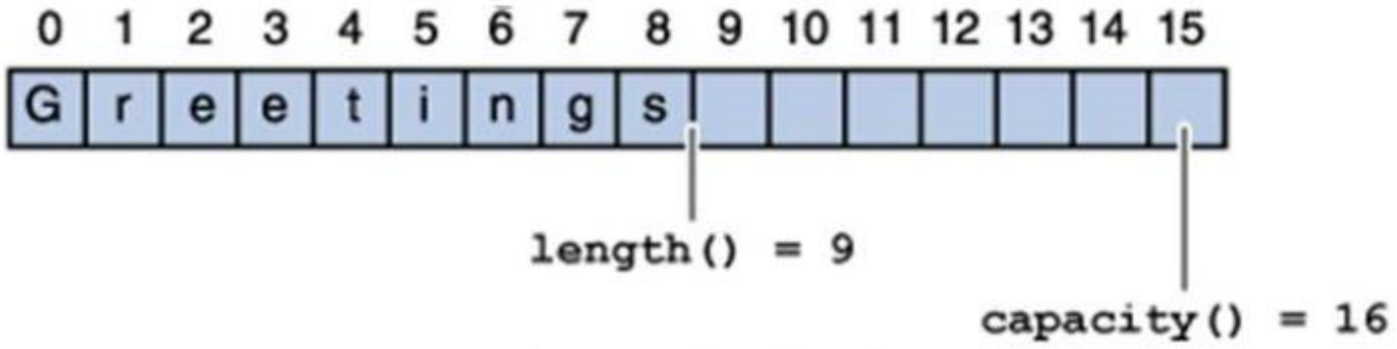
StringBuilder sb = new StringBuilder();

// creates empty builder, capacity 16

sb.append("Greetings");

// adds 9 character string at beginning

//will produce a string builder with a length of
//9 and a capacity of 16:



Length and Capacity Methods	
Method	**Description**
void setLength (int newLength)	Sets the length of the character sequence.
void ensureCapacity (int minCapacity)	Ensure that the capacity is at least equal to the specified minimum.

Various StringBuilder Methods	
Method	**Description**
StringBuilder append(double d) StringBuilder append(float f) StringBuilder append(int i) StringBuilder append(long lng) StringBuilder append(Object obj) StringBuilder append(String s)	Appends the argument to this string builder. The data is converted to a string before the append operation takes place.
StringBuilder delete (int start, int end) StringBuilder deleteCharAt (int index)	The first method deletes the subsequence from start to end-1 (inclusive) in theStringBuilder's char sequence. The second method deletes the character located at index.

StringBuilder Example

```
public class StringBuilderDemo {
   public static void main(String[] args)
    {
      String palindrome = "Dot saw I was Tod";
      StringBuilder sb = new StringBuilder(palindrome);
      sb.reverse(); // reverse it
      System.out.println(sb);
      }

      }
```

Understanding Pass by value and pass by reference in java

- Pass by value in java means passing a copy of the value to be passed.

- Pass by reference in java means the passing the address itself.

Pass by Values

- **Passing Primitive Variables:**

  When a primitive variable is passed the value with in the variable is copied into the method parameter, which is nothing but the pass by copy of the bits in the variable or pass by values.

```
Before Updating Value: 4
While Updating Value: 44
After Updating Value: 4
```

```
class PassPrimVar
{
    public static void main(String[] args)
    {
    int primitiveVariable = 4;
    PassPrimVar passObj =new PassPrimVar();
    System.out.println("Before Updating Value: "+ primitiveVariable);
    passObj.updateValue(primitiveVariable);
    System.out.println("After Updating Value:"+ primitiveVariable);
    }
    void updateValue(int var)
    {       var+=40;
            System.out.println("While Updating Value: "+var);
    }
}
```

Pass by Reference

- **Passing Object Reference Variables:**
- When a object variable is passed into the method, only the copy of the Object reference is passed and not the Object it self.
- The bit pattern in the reference variable is copied into the method parameter.
- What is this bit pattern? This bit pattern is the address of the specific object in the memory (on the heap).

```
import java.awt.Dimension;
class PassObjectRef
{
    public static void main(String[] args)
    {
        Dimension plotDimension = new Dimension(30,20);
        PassObjectRef  passObj=new  PassObjectRef();
        System.out.println("B4 Updating: "+"H:
        "+plotDimension.height+" W:"+plotDimension.width);
        passObj.updateDimension(plotDimension );


        System.out.println("After Updating:
      “+”  H:"+plotDimension.height+" W:"+plotDimension.width);

    }
```

```
        void updateDimension(Dimension dim)
        {
                dim.height+=10;
                //Notice the values being changed here.
                dim.width+=20;
        }
}
```

Output :

```
B4 Updating:    H: 30 W: 20
After Updating:  H: 40 W: 40
```

Wrapper Classes In Java

- Corresponding to every primitive data type there is a class which can encapsulate its value.

- These classes are known as wrapper classes

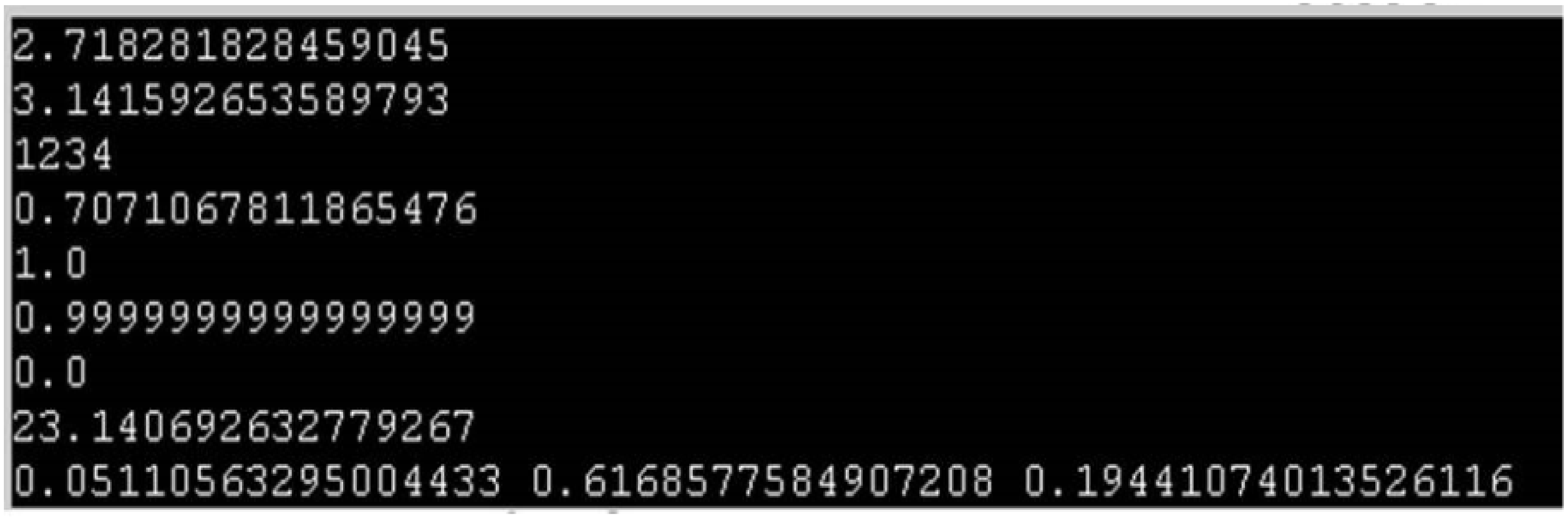
- All wrapper classes are final and immutable that means we cant create any sub class of the wrapper class and instance of wrapper cant be modified.

- Math **class** is declared final, so you cannot extend it. Constructor of Math **class** is private, so you cannot **create** an instance.

Methods and constants **from** Math **class** are static,

```
public class MainClass{
 public static void main(String args[]) {
        System.out.println(Math.E);
        System.out.println(Math.PI);
        System.out.println(Math.abs(-1234));
        System.out.println(Math.cos(Math.PI/4));
        System.out.println(Math.siN(Math.PI/2));
        System.out.println(Math.tan(Math.PI/4));
        System.out.println(Math.log(1));
        System.out.P.rintln(Math.exp(Math.PI));
  for(int i=0; i<3;++i)
        System.out.print(Math.random()+" ");
        System.out.println();
        }
        }
```

11



```
2.718281828459045
3.141592653589793
1234
0.7071067811865476
1.0
0.9999999999999999
0.0
23.140692632779267
0.0511056329500443 3 0.6168577584907208 0.1944107401352611 6
```