# CHAPTER 3 : PACKAGE

## What is a Package ?

Package is a collection of related classes and interfaces that provides access protection and namespace management. Generally, we are creating the package to get the following functionality—

1. It provides an inheritance hierarchy, so that we and other programmer are able to find the related classes and interfaces.
2. It provides unlimited access to each other within the package, still it provides restricted access to the outsiders of the package.
3. The name of a class or interface within a package will not conflict with the name of class or interface available in other package, because package namespace collision.

## HOW TO CREATE A PACKAGE?

A package is nothing but a folder in java that contains some classes and interfaces, so that it can be used by other classes like a header file.

Let us create a folder called geometry under D:\demo folder which will contain a folder called geometry that will contain Shape interface, Triangle and Rectangle class.

A source file which will remain in a package must provide the package statement as the first executable statement within itself as given below—

*package packagename;*

where, package is the keyword to create a package, and packagename is the user defined folder name.

ex- package geometry; → this statement must be the first executable statement of rectangle, triangle and shape.

## Creating a package called geometry :

```
/* Store this file as Shape.java under geometry folder */
package  geometry;
public  interface  Shape {
  void calArea();
}
/* Store this file as Rectangle.java under geometry folder */
package  geometry;
public  class  Rectangle  implements  Shape {
int  len , brdth;
Rectangle() {  len = 6;  brdth = 9;  }
public Rectangle ( int  p,  int  q )  {   len = p;   brdth = q;  }
public  viod  calArea()  {
     System.out.println("area of rec:"+len*brdth)}
 int   recArea(){
     return  len*brdth;
}};

/*  Store this file as Tringle.java under geometry folder */
package  geometry;
public class Triangle implements Shape {
int  ht, base;
Triangle () {  ht=5;  base=12;  }
public  Tringle(int p,  int  q) {  ht = p;  base = q;  }
public void calArea() {
        System.out.println("Aea of trg : " + ( ht*base ) / 2 );
}};
```

The above source files should be compiled in order to create/use the package. If we don't compile the source file then we cannot use the package in other file.

**Compiling the Source files of the Package :-** If there is any inheritance among the classes & interfaces of the package then we cannot compile the source file individually from the package itself. Hence we have to compile the source file at a time, as given below ..

```
D:\demo\geometry> javac *java
```

Irrespective of the inheritance relationship among the classes and interface of the package we can always compile the source file of the package individually or at once from the parent of the package.

```
d:\demo>   javac   d:\demo\geometry\Rectangle.java
                    (absolute path)
d:\demo>   javac   .\geometry\Triangle.java
                    (relative path)
```

**Using the geometry package :--** A source file which will use a package must provide the following import statement as the first statement before the class declaration.

< import packagename.***; > ->** it will include all dotclass files

```
Ex..  import  geometry.*;
      import java.util.*;
Or    < import packagename.dotclassfilename; >  -> It will include one dotclass file of the package
Ex    import   geometry.Rectangle;
       import   java.util.Scanner;
```

**Using the geometry package :**

Lets create a class test.java which will use the package

```
import geometry.*;
class  Test {
public static void main(String  args[]){
        Rectangle rec = new   Rectangle( 10, 12);
        Triangle trg = new  Triangle( 8, 10);
        rec.calArea();
        trg.calArea();
}};
```

The above source file should not be stored on the geometry folder otherwise compilation error will occur. Lets store Test.java under different folder as given bellow.

**Case->1 :** When implementation file (Test.java) is stored under the parent of the package(D:\demo), then it should be compiled and execute like a simple java program.

**Case->2 :** If the implementation file (Test.java) is stored under any folder except the parent of the package, then we have to compile and execute the implementation of file as given bellow (lets stored Test.java under e:\help)

```
Compile as:-   e:\help> javac  –classpath  d:\demo;.  Test.java
Execute as:-   e:\help> java   –classpath  d:\demo;.  Test
```

**classpath :** It is an enviornment variable because application program uses this variable. It is used to hold the path of a package ,zip file, jar file, war file etc. so that the application program will use this variable to find the path of the desire file

**-classpath :** It is an option for the java compiler and interpreter, when they are unable to find the path of a package then this temporary path is used.

**D:\demo : -** It is the parent of the package

**; (semicolon) :-** It is the path separator between different path.

**. (dot) :-** it represent the current folder value so that the compiler and interpreter will remember from where they have started compilation and execution?

**Case->3 :** If we set the permanent classpath of package, then we can compile & execute the implementation file like a simple java program. (Under the user variable list an environment variable dialog box look for classpath, if it is not available then click the new button, a dialog box will appear known as new user variable. Write **classpath** in the variable name and **.;d:\demo;** in the variable value click all ok or apply button after setting the permanent will can compile and execute Test.java like a simple java program .even if it is stored under any folder or drive.

**ACCESS SPECIFIER :-**A class may be preceded by public or default access specifier, when it default then it cannot be accessed by anybody outside of the package. When it is public then it can be accessed by anybody outside of the package.

| **Accessible of the members of class** | **Private** | **Default** | **protected** | **Public** |
|---|---|---|---|---|
| Within the same class | Yes | Yes | Yes | Yes |
| Within the same package & by subclass | No | Yes | Yes | Yes |
| Within the same package but by non subclass | No | Yes | Yes | Yes |
| Outside the package by subclass | No | No | Yes | Yes |
| Outside the package by non subclass | No | No | No | Yes |

**DEFAULT :-** A member of a class preceded by default access specifier can be access by any other class of the same package .Hence the default access specifier behaves like public for the other classes form the same package. The member of a class presided by default access specifier can't be accessed by any other class that remains outside of the package hence default behaves like private for the classes outside of the package.

**PROTECTED :-** The member of the class preceded by the protected access specifier can be access by any other class within the same package .Hence the protected access specifier can behaves like public. The member of a class preceded by protected access specifier can't be by any non sub class that remain outside of the of a package, hence protected behaves like private for the non subclasses that remains outside of the package. The member of a class preceded protected access specifier can be accessed by the subclass that remain outside of the package , even if the class is not allowed to access its own protected member outside of the package

The protected and default access specifier allows everybody within the package but they does not allows anybody to access outside of package where as the protected will allows the child class to access even if the child class is outside of the package hence package provides access protection.

**Private access specifier is called class scope, default access specifier is also called package scope.**

**Namespace Management :** The name of a class/interface along with the memory space/folder/package where it is stored is known as namespace. Java allows us to provide same name to different source files by storing them in different forder/package.

```
import   java.util.*;
import   java.sql.*;
class Test{
        static public void main(String args[]){
```

```
                        Date d = new Date();
        }}
```

The Date class in above program is available in util & in sql package. Since we are using both package, then compiler will face an ambiguity to create the Date object. It is known as namespace collision/conflict. Java avoids the namespace collision/conflict by using the long qualified name of the package as given bellow
java.util.Date d = new java.util.Date(); // java.util.Date is called long qualified name.

**<u>How To Create a Sub-package :</u>** The stpes describes how to create the subpackages,
**Stpe 1 :** Create a folder called drawing under geometry foldet, so that the drawing folder will behave like a sub package.
**Step 2 :** The drawing package should contain a class Square which will inherit from Rectangle class. Hence the default constructor and the Rectangle must be public or protected, because when the object of Square class will be created, then the default constructor of Rectangle class will be invoked implicitly.
**Step 3 :** Set the permanent classpath of main(geometry) package in environmental variable.
**Step 4 :** Since the source file that stored in a package contains the package statement as the 1st statement, hence the source file of drawing package i.e. Square class must provide package & sub package name as the package statement within it.

```
 package  geometry.drawing;
 public class Square  extends  geometry.Rectangle {
   int side;
   public Square(){ side=100;  }
   public Square(int p){ side=p;  }
   public void calArea(){
          System.out.println("area of Square "+ side* side);
}};
```

Store the above file as Square.java in drawing package,
**Step 5:** Compile the Square.java from the parent of main(geometry) package as given below.

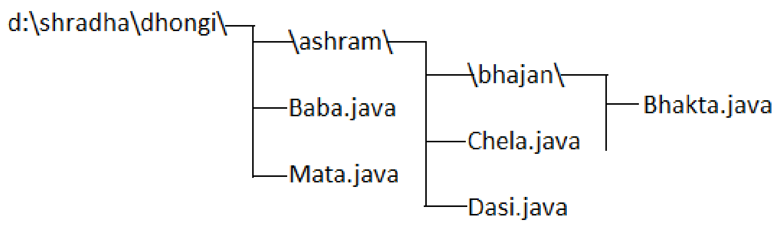
```
D:\demo> javac   .\geometry\drawing\Square.java
```

**Step 6 :** The implementation file(Test.java) should be written as given below :

```
import  geometry.*;
import  geometry.drawing.*;
class  Test   {
  public static void main(String args[]){
    Rectangle rec = new Rectangle(10,12);
    Triangle trg = new Triangle(8,10);
    rec.calArea();
    trg.calArea();
    Square sq = new  Square();
    sq.calArea();
}};
```

**Step 7 :** Since we have set the permanent classpath of the main(geometry) package, therefore , the Test.java can be stored any location and it can be compiled and executed like a simple java program.

**Creating another Sub Package :** Let us create a package called "dhongi" under d:\shradha folder and this package a sub package called "ashram" & ashram has a sub package a sub package called "bhajan", the package & sub package consisting of following classes :



Step -1 : No need to create the folder just store the following file directly under d:\shradha :

| | |
|---|---|
| //store as Baba.java under shradha folder<br>package dhongi;<br>public class Baba {<br> public Baba(){ }<br> public void thaka(){<br>  System.out.println("baba in thaka method");<br>}}; | //store as Mata.java under shradha folder<br>package dhongi;<br>public class Mata {<br> public Mata(){ }<br> public void mohini(){<br>  System.out.println("mata in mohini method");<br>}}; |
| //store as Chela.java under shradha folder<br>package dhongi.ashram;<br>public class Chela extends dhongi.Baba {<br> public Chela(){ }<br> public void thaka(){<br>  System.out.println("chela n baba in thaka method");<br>}}; | //store as Dasi.java under shradha folder<br>package dhongi.ashram;<br>public class Dasi extends dhongi.Mata {<br> public Dasi(){ }<br> public void mohini(){<br>  System.out.println("dasi n mata in mohini method");<br>}}; |

```
//store  as  Bhakta.java  under shradha folder
package  dhongi.ashram.bhajan;
public class  Bhakta extends  dhongi.ashram.Chela{
  public  Bhakta(){      }
  public  void   thaka(){
   System.out.println("bhakta  in  thaka method");
 }
 public void  bechara(){
 System.out.println("bhakta becomes bechara");
}};
```

Step-2 : Compile from the dos prompt as :

```
javac  -d  .  *.java
```

where, -d is a directory option to the compiler and it specifies where to place generated .class files.
. (dot) represents current folder, so that .class file will be stored in current folder

Step-3: Copy the source file, i.e. all *.java file to a test folder within d:\shradha
Step-4: Create an implementation file, say Human.java under d:\shradha folder which will use all the package & sub package.

```
import  dhongi.*;
import  dhongi.ashram.*;
import  dhongi.ashram.bhajan.*;
class Human {
public static void main(String args[]) {
Baba b1= new  Baba();
b1.thaka();
Mata m1= new Mata();
m1.mohini();
Chela c1=new Chela();
c1.thaka();
Dasi  d1= new  Dasi();
d1.mohini();
Bhakta  bh1=new Bhakta();
bh1.thaka();
bh1.bechara();
}};
```

Step-4:
Compile & execute this program.

**Note :** The following points should be remembered about a package—

1. The package name in package statement and in import statement must remain in similar case, otherwise compilation error will occur.
2. When we import a package, then we can use the classes and interfaces of the package , but we cannot use the classes and interfaces of the sub-package. Hence to use the classes and interfaces of the sub-package we have to import the sub-package.
3. We can use a class or interface of a package without import statement , but we have to use long qualified name of the package.

**HEADER FILE Vs PACKAGE :**

1. A package provides security by access specifiers where as header file of c does not provide any security but header files of c++ provides the security.
2. Package contain sub-packages, but header files of c and c++ does not provide sub-header files.
3. Package manages the namespace collision, but header files of c manages it by using Macros and header files of C++ manages it by using predefined namespace(std) along with user-defined namespace.

**#include vs import :-**

**#include –** It copies the code available in header file in a user defined C program. Hence it takes more time to compile a C program but it takes to load and link, in turn the execution becomes faster.

**Import –** It creates a link between the method invoke by the programmer from the java source file to the method defined in the package, hence java take less time to compile. But execution becomes slower because control jumps from method convocation to method definition.

**java.lang package :--**This is known as java language package. It is the default package of java, because it is implicitly available to each and every java source file we are defining. Hence no need to import the package. The classes and interfaces of the lang package are as follows :

| CLASSES | | | | INTERFACES |
|---|---|---|---|---|
| String | Process | Byte | | Runnable |
| StringBuffer | ProcessBuilder | Short | | Cloneable |
| Thread | Runtime | Integer | | Comparable |
| Throwable | SecurityManager | Float | Wrapper class | |
| Object | Math | Long | | |
| Runtime | System | Double | | |
| Class | | Boolean | | |
| | | Character | | |

**Object class :**

It is the base or root class of java programming language because from it all other classes are derived, hence it is known as cosmic class.
When a class Z will not inherit from any other class, then the Object class will become the parent class of Z. When a class Z will inherit from any other class then the Object class will not become the parent class of Z. Therefore, the Object class is the default parent class of java programming language.

**Printing an object :-**

```
class  A  {
     int x,  y;
     A(){  x= 5;  y=12;   }
    A(int  p,  int  q) {     x=p;    y=q;  }
    public  String  toString() {
        return  x + " : " + y;
 }};

class  B {
  public static void main(String  args[]) {
  int  p = 234;
  System.out.println(" the value : " + p );
  A   a1  =  new  A();
  System.out.println( a1);
  System.out.println( a1.toString());
}};
```

**public String toString ()**-- This method of Object class is used to print the value of data members of an object like a String. We can say this method is used to return the state of an object as a string.
Whenever we pass an object of any class to the println() method, then implicitly the interpreter will invoke the toString() on the object of any class to print the value of data member.

If we do not override the toString() in our class then the toString() of Object class will be invoked only when we pass the object of our class i.e. a1 to the println() and that will produce the following value—

```
                A@1a46e30
```

The above output will be displayed because we have not overridden the toString() method in the our Stud class, and our class will get the following toString() method of the Object class :

```
public  String  toString(){
  return getClass().getName()+ "@" + Integer.toHexString(hashcode());
  //return  classname@hexa_decimal_String_representation_of-hashcode;
}
```

where hexcode is the hexadecimal no. used by the interpreter to get the memory location of an object.

**Runtime class(Playing Video, music) :** —It is a singleton class, hence its object cannot be created by the constructor. Therefore, the static method called getRuntime() is used to return an object of this class. The exec() method of this class is used to execute or initiate any application i.e. .exe file from the java program.

```
class  Runexe  {
public static void main(String  args[]) throws   Exception {
    Runtime rt = Runtime.getRuntime();
    Process  p1 = rt.exec("free cell");
    Process  p2 = rt.exec("calc");
    Process p3= rt.exec("C:\Program Files (x86)\VideoLAN\VLC\vlc.exe  aa.mp4");
    Thread.sleep(5000);
    p3.destroy();
}};
```

**Process Class :** A program in execution is known as process, hence whatever .exe file we are executing by exec() method can be refereed by Process class in the java program, because the return type of exec() is Process class.

**Q : Compile Y.java during the execution of X.java**

**Store it as Y.java :**

```
class Y{  };
```

**Store it as X.java :**

```
class X {
 public static void main(String args[]) throws Exception {
  Runtime rt = Runtime.getRuntime();
  Process p1=rt.exec("javac  Y.java");
}};
```

**Note :-**Each and every object is stored in heap area. The heap area is divided into different pools.

**1) Referenced pool :** In the referenced pool all the objects references maintained.

**2) Object pool :** In this pool all the objects are located here by using new operator.

**3) Thread pool :** In java we are able to create a thread object which not contains any lock

**4) String Contant Pool :** Without using new operator we are creating objects, i.e. by sString literal, such type of the objects are located here(ex:-String object).

**String class :** String is the final class in java. Hence it cannot be inherited. String objects are immutable i.e. once the object is created the contents at the memory location cannot be changed.

**SCP:** Java reserved some part of heap are for String object, this area is called **String Constant Pool(SCP).**

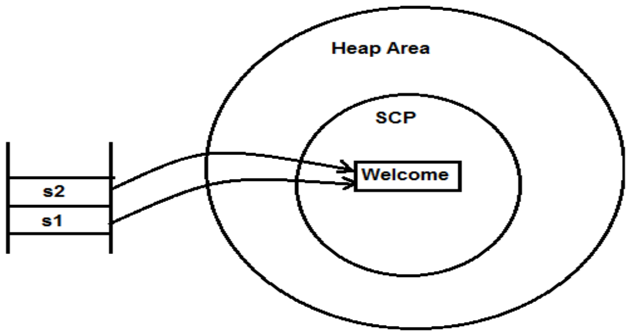
1) String is a predefined final class it is present in java.lang package.
2) It can not be inherited.
3) Generally, string is a sequence of characters. But in java, string is an object that represents a sequence of characters. String class is used to create string object
4) Once we are creating String object it is not possible to do the modifications on existing object called immutability nature.
5) It has some predefined constructor and predefined method.
6) The constructor of String class is overload.
   How to create String object
   There are two ways to create String object:
   1. By string literal
   2. By new keyword

**Note:**

1) String objects are stored in a special memory area known as string constant pool.If a string object is created directly, using assignment operator.String s1="hi"; then it is stored In string constant pool
2) The Object present in the SCP is not eligible for garbage collection even though the object doesn't have any reference variables.All the SCP objects will be destroyed at the time of JVM shutdown.
3) In the scp there is no chance of 2 string objects with the same content i.e duplicate string objects won't present in scp.

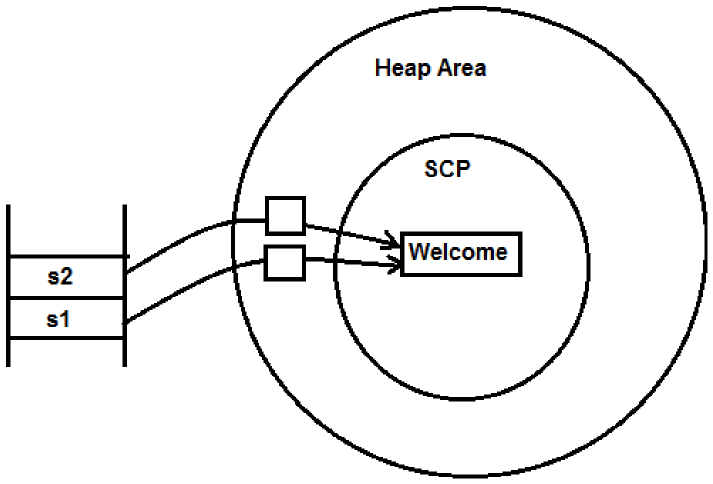
**1. Creating String Literal :** Java String literal is created by using double quotes.

Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference of the pooled instance is returned. If string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

| | |
|---|---|
| String s1 ="welcome";<br>String s2 ="welcome";<br>//will not create new instance |  |

2. **Creating String object by new & constructor :** Java String object can be created by using new & constructor. For Example :

```
String s1= new  String("rahu");
String s2= new  String("rahu");
```

| | |
|---|---|
| Each time you create a string object(say s1) by using constructor, the JVM allocates a memory in heap area, then the string literal "rahu" given in the constructor will be allocated in SCP, only if "rahu" is not present in SCP, then a address of literal "rahu" present in SCP will be stored in memory allocted for object(s1) in heap are. The object(s1) in the program will get the address of heap area. When we create another object(s2) by using constructor then JVM allocates another memory in heap area, but this object has the same literal "rahu", hence the memory address of "rahu" present in SCP during 1st object(s1) creation will be returned to the memory allotted in heap area for s2, and the object(s2) in the program will get memory address of heap area. |  |

Hence, for each String object created by new & constructor JVM creates 2 memory location, i.e. 2 objects.

**Immutability property of String object.**

```
class  Str1  {
public static void main(String  args[]) {
  String  s1 = "java" ;
  String  s2 =  s1;
  System.out.println(s1);
  System.out.println( s1 );
  System.out.println( s2 );
  s1 = s1  + "  program";
  System.out.println( s1 );
  System.out.println( s2 );
}};
```

**public boolean equals(Object ob) :** This method of Object class is used to compare two objects. It is inherited to String class. If the content of both string object are equal and in same case then it will return true otherwise false.

**Equality operator(==) :** This operator will evaluate only string objects. If the content and memory location of both the string objects are equal then it will return true, otherwise it will return false.

```
class  StringDemo1 {
  public  static void main(String args[]) {
       String s1 = "java";
       String s2 = "java";
       System.out.println( s1.equals(s2) );
       System.out.println( s1 == s2 );
       String s3 = new String( "java");
       String s4 = new String("java");
       System.out.println( s3.equals(s4) );
       System.out.println( s3 == s4 );
}};

class  StringDemo2 {
  public  static void main(String args[]) {
       String s1 = "Learning string manipulation";
       System.out.println( s1.charAt(5) );
       System.out.println( s1.indexOf('t') );
       String s2[] = s1.split(" ");
       for ( int i= 0 ;  i < s2.length ; i++)
          System.out.print( s2[i] +"  ");
}};
```

**<u>Method of  String  class :-</u>**

1) char charAt(int index)
2) int compareTo(String ob)
3) boolean endsWith(String suffix)
4) boolean equalsIgnoreCase(String ob)
5) int indexOf(char ch)
6) int indexOf(String str)
7) int indexOf(char  c, int beginIndex)
8) boolean isEmpty()
9) int lastIndexOf(int ch)
10) int lastIndexOf(String str)
11) int lastIndexOf(String str, int   fromIndex)
12 ) int length()
13) boolean startsWith(String prefix)
14) String substring(int beginIndex)
15) String substring(int beginposition, int  endindex)
16) String   toLowerCase()
17) String   toUpperCase()
18) String    trim()
19) String[]  split(String ob)

Q) Wap to write a program to perform the addition of two big numbers when the numbers cannt be stored in data types.?

**StringBuffer Class :**

1) String Buffer is a class present in the java.lang package.
2) StringBuffer is a final class so it can't be inharited.
3) StringBuffer is a mutable class , i.e. it is possible to change the content in the same location.
4) StringBuffer .equals() method is used for reference comparision.
5) Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously.

**Constructors:-**

1) StringBuffer sb=new StringBuffer(); StringBuffer object will have a capacity of 16 characters.
2) StringBuffer sb=new StringBuffer(String str);
3) StringBuffer sb=new StringBuffer(int capacity); It is possible to store more than 30 characters into StringBuffer.

To store characters, we can use append () method as:     sb1.append ("Kiran");

**Methods of StringBuffer :**

1. public synchronized StringBuffer append(String s): is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
2. public synchronized StringBuffer insert(int offset, String s): is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
3. public synchronized StringBuffer replace(int startIndex, int endIndex, String str): is used to replace the string from specified startIndex and endIndex.
4. public synchronized StringBuffer delete(int startIndex, int endIndex): is used to delete the string from specified startIndex and endIndex.
5. public synchronized StringBuffer reverse(): is used to reverse the string.
6. public int capacity(): is used to return the current capacity.

7. public void ensureCapacity(int minCapacity): is used to ensure the capacity at least equal to the given minimum.
8. public char charAt(int index): is used to return the character at the specified position.
9. public int length(): is used to return the length of the string i.e. total number of characters.
10. public String substring(int beginIndex): is used to return the substring from the specified beginIndex.
11. public String substring(int beginIndex, int endIndex): is used to return the substring from the specified beginIndex and endIndex.

**What is mutable string**

A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

## StringBuilder Class :

It is a predefined class in java.lang package can be used to handle the String. StringBuilder class is almost similar to to StringBuffer class. It is also a mutable object. The main difference StringBuffer and StringBuilder class is StringBuffer is thread safe that means only one threads allowed at a time to work on the String where as StringBuilder is not thread safe that means multiple threads can work on same String value.

**Constructors :-**

1. StringBuilder(): creates an empty string Builder with the initial capacity of 16.
2. StringBuilder(String str): creates a string Builder with the specified string
3. StringBuilder(int length): creates an empty string Builder with the specified capacity as length.

**When we use String, StringBuffer and StringBuilder**

- If the content is fixed and would not change frequently then we use String.
- If content is not fixed and keep on changing but thread safety is required then we use StringBuffer, generally used ofr uodation purpose.
- If content is not fixed and keep on changing and thread safety is not required then we use StringBuilder , generally use for simultaneous access of result, scores, etc.
-

| | StringBuffer | StringBuilder |
|---|---|---|
| 1 | It is thread safe. | It is not thread safe. |
| 2 | Its methods are synchronized and provide thread safety. | Its methods are not synchronized and unable to provide thread safety. |
| 3 | Relatively performance is low because thread need to wait until previous process is complete. | Relatively performance is high because no need to wait any thread it allows multiple thread at a time. |
| 1 | Introduced in 1.0 version. | Introduced in 1.5 version. |

| | String | StringBuffer | StringBuilder |
|---|---|---|---|
| **Storage Area** | String Constant Pool | Heap | Heap |
| **Modifiable** | No (immutable) | Yes( mutable ) | Yes( mutable ) |
| **Thread Safe** | Yes | Yes | No |
| **Performance** | Fast | Very Slow | Fast |

**How to create Immutable class?** There are many immutable classes like String, Boolean, Byte, Short, Integer, Long, Float, Double etc. In short, all the wrapper classes and String class is immutable. We can also create immutable class by creating final class that have final data members as the example given below:

```
public final class Emp{
        final String  panNumber;
         public Emp(String pan){
                  this.panNumber=panr;
        }
        public String getPanNumber(){
                return  panNumber;
}}
```

**Using java documentation/help/api( application programming interface) : --**

1) Extract jdk-8-doc.zip to d:\
2) A folder called docs will be created under d:\
3) Double click on index.html file of d:\docs\api folder, it will open a html file consisting of jdk API.

| **New from jdk1.5 :--** | |
|---|---|
| 1) varargs | 5) static import |
| 2) for-each loop(new for loop) | 6) generic/templates |
| 3) printf() | 7) enums |
| 4) autoboxing/autounboxing | 8) instanceof |

**Autoboxing :**

Upto java 1.4 the predefined datatype value is converted into corresponding wrapper class object by passing the datatype value in the constructor of the wrapper class as given below---

```
Integer ob1=new Integer(12);
Float ob2= new Float(23.33F);
```

From java 1.5, a new concept called autoboxing is used to convert the value of datatype into corresponding wrapper class object as given below-

```
Integer ob1= 12;
Float ob2= 23.33F;
```

The process by which java converts predefined datatype value into corresponding wrapper class object is called autoboxing.

**Autounboxing :**

Upto java 1.4, to convert the value of wrapper class object into corresponding datatype value we have to use the methods of wrapper classes as given below-

```
int y= ob1.intValue();
float z= ob2.floatValue();
```

From java 1.5, a new concept called autounboxing is used to convert the value of wrapper class object into corresponding datatype as given below

```
int y= ob1;
float z= ob1;
```

The process by which the wrapper class object is converted into corresponding data type value is called autounboxing

**Static import :**

If we write the static keyword in between the import keyword and the package name, then it is called static import. It is used to import the static members of a class, so that we can use the static members in our class as if we have defined it.

When we import the static members of a class by static import, then we should not define any member in our class with the same name as that of the static import.

```
import  static  java.lang.Math.pow;
import  static  java.lang.Math.sqrt;
import  static  java.lang.System.out;
class  StatImp {
  public static void main(String args[]){
    //without static import
```

```
   double  x = Math.sqrt( Math.pow(4,2) + Math.pow(3,2) );
   System.out.println(x);

   //with static import
   double y= sqrt(pow(8,2) + pow(6,2) );
   out.println(y);
}};
```

**instanceof operator :** This operator decides wheather an object belongs to an inheritance hierarchy or not. If an object belongs to a class or the object can be hold by any of it's prent class then instanceof returns true. When the object of a class does not belong to it's child class then it will return false. If the object does not belong to an inheritance hierarchy then it will generate compilation error.

```
class A  {   }
class  B  extends  A{  }
class  C  extends  B{  }
class  D  extends  A{  }
class  StatImp {
  public static void main(String args[]){
    A   a1 = new  A();
    System.out.println(a1  instanceof  A);
    System.out.println(a1  instanceof  B);
    System.out.println(a1  instanceof  C);
    System.out.println(a1  instanceof  D);

    B   b1 = new  B();
    System.out.println(b1  instanceof  A);
    System.out.println(b1  instanceof  B);
```

```
    System.out.println(b1  instanceof  C);
//    System.out.println(b1  instanceof  D);

    C   c1 = new  C();
    System.out.println(c1  instanceof  A);
    System.out.println(c1  instanceof  B);
    System.out.println(c1  instanceof  C);
//    System.out.println(c1  instanceof  D);

    D   d1 = new  D();
    System.out.println(d1  instanceof  A);
    System.out.println(d1  instanceof  B);
//    System.out.println(d1  instanceof  C);
    System.out.println(d1  instanceof  D);
}};
```

## Methods of Object class :

**public String toString()** : If you want to represent any object as a string, toString() method is used. The toString() method returns the string representation of the object.

**Q).** If we pass an object of ArithmeticException class then we will get ArithmeticException, describe, how the toString() is overridden.

**Q).** C is a child of B, B is a child of A, and by passing object of C in Sopl() we get A-B-C, but by passing object of B we get A-B, how the toString() is overridden and in which class, can we reuse it.

**public int hashCode() :**

For every object, JVM will generate an unique no. which is nothing but hashCode(). Internally JVM will generate hash code based on address, but it does not mean, hash code represent address of an object.

JVM will use that hash code while saving objects into hashing related data structures like HashSet, Hashtable, HashMap.The main advantage of saving object based on hash code is search operation will become efficient( hashing is the most powerful search algorithm and its time complexity is 0(1).

Overriding hashCose() is said to be proper if and only if we have to generate an unique no. as hash code for every object.

| | |
|---|---|
| class Stud{<br>  public int hashCode() {<br>    return 100;<br>}};<br>This is improper way of overriding hashCode(), because for all object of Stud class  we are generating same hash code. | class Stud{<br>  public int hashCode() {<br>    return rollno;<br>}};<br>This is proper way of overriding hashCode(), because for every object of Stud class  we are generating unique no. as hash code. |

**toString() vs hashCode() :** If we override toString() in our class then toString() will not call the hashCode(), but if we do not override the toString() in our class then predefined toString() of Object class will be available in our class, and it will call the hashCode() of the Object class internally.

```
class A{
 int x;
 A(int a){ x=a;   }
};
```

```
class B{
   int x;
   B(int a){  x=a;    }
   public String toString(){
       return "object of B : "+x;
 }};
```

```
class C {
 int x;
 C(int a){  x=a;    }
 public  int hashCode(){    return x;  }
 public String toString(){   return "object of C : "+x;
 }};
```

```
class D {
public static void main(String args[]){
   A  a1=new A(10);     System.out.println(a1+", "+Integer.toHexString(a1.hashCode()));
   B  b1=new B(10);     System.out.println(b1+", "+b1.hashCode());
   C  c1=new C(10);     System.out.println(c1+", "+c1.hashCode());
}}
```

**public boolean equals() :** This method of the object class is used for reference comparison but for String class, Wrapper class and all Collection class it is used for content comparison.

```
String s1= new String("abc");
String s2= new String("abc");
Sopl(s1.equals(s2)); //true
```

```
StringBuffer sb1= new StringBuffer("abc");
StringBuffer sb2= new StringBuffer("abc");
Sopl(sb1.equals(sb2)); //false
```

Based on our requirement we can override the equals() for content comparison. Whenever we are overriding equals(), we have to consider the following contact of equals() :

i) The meaning of content comparison is that whether we check only one data member or all data member( like in Stud class whether we like to check only name or name & roll)

ii) If we pass object of different class(heterogeneous object) to equals(), then we should handle the ClassCastException to return false.

iii) If we pass a reference to equals() , then we should handel NullPointerException to return false.

Now, keeping above contract in our mind let'us define equals() in our class.

```
class  Stud{
   String name;
   int roll;
   Stud(String a, int b){
         this.name=a;   this.roll=b;}
   public boolean equals(Object o){
   try{
      String na= this.name;
      int ro=this.roll;
      Stud s=(Stud) o;
      String na1=s.name;
      int ro1=s.roll;
//    if( na.equals(na1) && ro == ro1)
  if(   name.equals(s.name) && roll  ==
s.roll )
         return true;
  else
        return false;
  }catch(ClassCastException e){
        return false;
  }catch(NullPointerException e){
       return false;
  }
}};
```

```
class  Cell{
 String name;
 double price;
 Cell(String a, double p){name=a; price=p; }
};

class  Eqtest {
public static void main(String args[]){
Stud s1= new Stud("suji", 101);
Stud s2= new Stud("washim", 102);
Stud s3= new Stud("suji", 101);
System.out.println(s1.equals(s2));
System.out.println(s1.equals(s3));
System.out.println(s2.equals(s3));
System.out.println(s2.equals("wasim"));
System.out.println(s1.equals(null));

Cell c1= new Cell("apple", 2000.00);
Cell c2= new Cell("lemon", 2000.00);
Cell c3= new Cell("apple", 2000.00);
System.out.println(c1.equals(c2));
System.out.println(c1.equals(c3));
System.out.println(c2.equals(c3));
}};
```

**protected Object clone() throws CloneNotSupportedException :** The process of creating exactly duplicate object is called cloning, we do not need to call the new operator with the constructor to create another object. The main objective of cloning is to maintain backup purpose. We can performing cloning by using clone() method of Object class.

We can perform cloning only for cloneable objects, an object is said to be cloneable if and only if the corresponding class implements the predefined Cloneable interface present in java.lang package. This interface does not contain any methods, it is a marker interface. If we try to use clone() method of Object class on non-cloneable objects then we will get CloneNotSupportedException.

Cloning are of 2 types known as Swallow Cloning/Copy and Deep Cloning/Copy.

**Swallow Cloning/Copy**

**:** The process of creating bitwise copy of an object is called swallow cloning. A new object is created that has an exact copy of the values in the original object(without calling the constructor with new operator). If any of the fields(data member/instance variable) of the original object is a reference variable, then the cloned object does not have a duplicate reference variable, rather the cloned object will have a reference variable which will point to the memory location of reference variable of original object, but if the fields are of primitive/String/Wrapper type then the duplicate copies will be created in cloned object. By using the main object if we change the value of reference variable, then the changes will be reflected in the cloned object and vice versa.

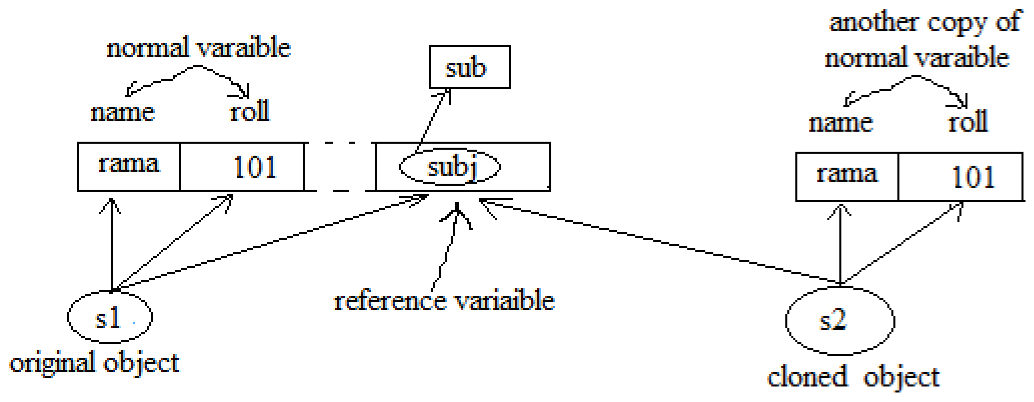
```
class Subject {
  String  sub;
  Subject(String s) {    sub = s;  }
  public String toString(){
      return sub;
}};
class Stud implements Cloneable {
  String  name;
  Subject  subj;//Contained object
  int  roll;
  Stud(String n, Subject s, int r) {
    name = n;  ubj=s;   roll=r;
  }
  public Object clone() {    //shallow copy
          return super.clone();
}};
class swalclone {
public static void main(String[] args) {
Subject sb1= new Subject("Java");
Stud s1 = new Stud("Rama", sb1,101);
System.out.println("Original : " + s1.name + " - "+ s1.roll + " - "+ s1.subj);

Stud s2 = (Stud) s1.clone();     //Cloned Object
System.out.println("Cloned : " + s2.name+ " - "+ s2.roll+ " - "+ s2.subj);

s1.name="Sama";
s1.subj.sub="Adv Java";
s1.roll=420;
System.out.println("Original after update: " + s1.name + " - "+ s1.roll + " -
"+ s1.subj);
System.out.println("Cloned after updation: "+ s2.name + " - "+ s2.roll + " -
"+ s2.subj);
}};
```

**Output :**

```
Original : Rama - 101 - Java
Cloned : Rama - 101 - Java
Original after updation: Sama - 420 - Adv Java
Cloned after updation: Rama - 101 - Adv Java
```



now if we make changes to the refered object, then both
s1 and s2 will get the same value of the subject

To avoid the above problem deep cloning is used. Swallow cloning is best useful if the object contains only primitive/String/Wrapper values.

**Deep Cloning/Copy :** The process of creating exactly duplicate independent object including reference variable i.e. contained object is called deep cloning. In this case we have 2 different memory location for the primitive/String/Wrapper values as well as for reference variable(i.e contained object.
The clone() method of the object class is meant for swallow cloning, if we want deep cloning, then programmer is responsible to override clone() method by writing own logic.
In deep cloning by using the main object if we change the value of reference variable then the changes will not be reflected in the coned object.

```
class  Tiger{
String  color;
int wt;
Tiger(String c, int w) {  color=c;  wt=w; }
public String toString(){
   return color + " : "+wt;
}};

class  Zoo implements Cloneable {
  //Contained object
  String  name;
  Tiger trg;
  Zoo(String n, Tiger t) {  name=n;  trg=t;  }
  public Object clone() {
   //deep copy
     Tiger tt= new Tiger(trg.color, trg.wt);
      Zoo  ob = new Zoo(name, tt);
      return ob;
}};
```

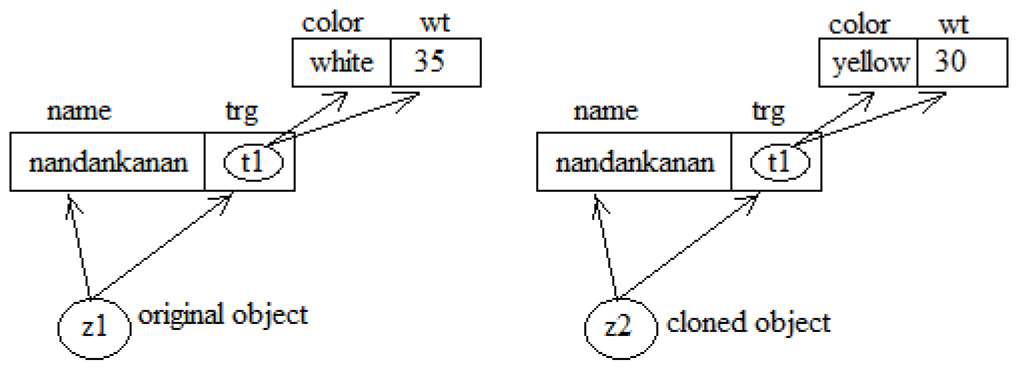
```
class deepclone {
public static void main(String[] args) {
Tiger t1 = new Tiger("white", 35);
Zoo  z1=new Zoo("nandankanan",t1);   //Original Object
System.out.println("Original object: " + z1.name + ", "+
z1.trg);

Zoo  z2=(Zoo) z1.clone();      //Clone Object
System.out.println("Cloned object : " + z2.name + ", "+
z2.trg);

z2.name="Zoopark";
z2.trg.color="yellow";
z2.trg.wt=30;
System.out.println("Original after change: " + z1.name
+ ", "+ z1.trg);
System.out.println("Clone after change: " + z2.name +
", "+ z2.trg);
}};
```

**Output:**

```
Original object: nandankanan, white : 35
Cloned object : nandankanan, white : 35
Original after change: nandankanan, white : 35
Clone after change: Zoopark, yellow : 30
```



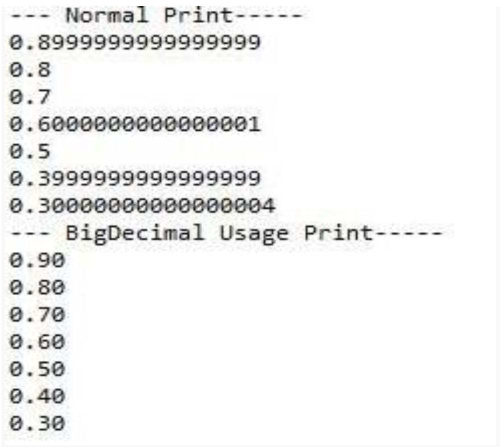
after creating cloned object, if we change in reference variable trg then it will not be reflected in other object

**Java BigDecimal Class :** Java includes BigDecimal class for performing high-precision arithmetic which can be used in banking or financial domain based application. This class approximately fit into the same category as the "wrapper" classes but has some very useful methods.

This class has methods that provide analogues for the operations that you perform on primitive types. That is, you can do anything with a BigDecimal that you can with an int or float, it's just that you must use method calls instead of operators. Also, since there's more involved, the operations will be slower. You're exchanging speed for accuracy.

BigDecimal is for arbitrary-precision fixed-point numbers; you can use these for accurate monetary calculations. Below Java code explain the concept of accuracy in calculation. One part is doing all calculation using double while other part is having calculation using BigDecimal. Output shows difference between them.

```
import java.math.BigDecimal;
public class BigDecimalDemo {
public static void main(String[] argv) {
        System.out.println("--- Normal Print-----");
        System.out.println(2.00 - 1.1);
        System.out.println(2.00 - 1.2);
        System.out.println(2.00 - 1.3);
        System.out.println(2.00 - 1.4);
        System.out.println(2.00 - 1.5);
        System.out.println(2.00 - 1.6);
        System.out.println(2.00 - 1.7);
        System.out.println("--- BigDecimal Usage Print-----");
        System.out.println(new BigDecimal("2.00").subtract(new BigDecimal("1.1")));
        System.out.println(new BigDecimal("2.00").subtract(new BigDecimal("1.2")));
        System.out.println(new BigDecimal("2.00").subtract(new BigDecimal("1.3")));
        System.out.println(new BigDecimal("2.00").subtract(new BigDecimal("1.4")));
        System.out.println(new BigDecimal("2.00").subtract(new BigDecimal("1.5")));
        System.out.println(new BigDecimal("2.00").subtract(new BigDecimal("1.6")));
        System.out.println(new BigDecimal("2.00").subtract(new BigDecimal("1.7")));
        BigDecimal bd1 = new BigDecimal ("1234.34567");
        bd1= bd1.setScale (3, BigDecimal.ROUND_CEILING);
        System.out.println(bd1);
}}
```



```
--- Normal Print-----
0.8999999999999999
0.8
0.7
0.6000000000000001
0.5
0.3999999999999999
0.30000000000000004
--- BigDecimal Usage Print-----
0.90
0.80
0.70
0.60
0.50
0.40
0.30
```

**Limiting a number to a certain degree of precision** : Let us say, we would like to calculate something, up to 3 decimal places, no more and no less, and independent of the number we used to instantiate the BigDecimal. In that case, we will have to specify two properties of a BigDecimal object. The rounding mode and the scale. The scale defines the number of decimal places we want, and the rounding mode specifies how we would like to round a floating point number. As an example :

```
BigDecimal bd1 = new BigDecimal ("1234.34567");
bd1= bd1.setScale (3, BigDecimal.ROUND_CEILING);
System.out.println(bd1);    Output would be 1234.346
```

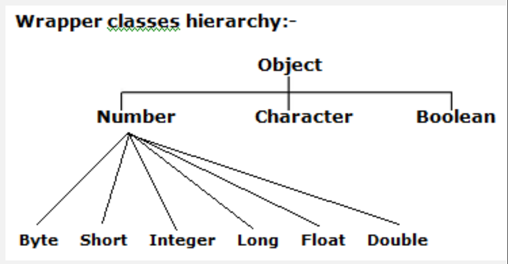
**Q) What is "enum" , Enum and Enumeration :**
**"enum"** : It is a keyword which can be used to define a group of normal constant.

**Enum** : It is a class present in java.lang package. Every enum in java is the direct child class of Enum class, hence the Enum class acts as base class for all Java enums.

**Enumeration** : It is an interface present in java.util package. We can use Enumeration interface to get object one by one from collection.

**WRAPPER CLASS :** The main objectives of wrapper classes are:

1. To Wrap primitives into object form. So that we can handle primitives also just like objects.
2. To represent primitive data types as a Object form we required some classes these classes are called wrapper classes.
3. All wrapper classes present in the java.lang package.
4. Int ,byte…. Acts as a primitives we can make the primitives into the objects is called wrapper classes the the wrapper classes are Integer,Byte-----.
5. **5.** We are having 8 primitive data types hence sun peoples are providing 8 wrapper classes.

| Primitives | Wrapper classes | Fallowing constructor arguments | |
|---|---|---|---|
| Byte | Byte | Byte or String |   Wrapper classes hierarchy:-<br>Object<br>Number<br>Character<br>Boolean<br>Byte<br>Short<br>Integer<br>Long<br>Float<br>Double |
| Short | Short | Short or String | |
| Int | Integer | Int or String | |
| Long | Long | Long or String | |
| Float | Float | Float or double or String | |
| Double | Double | double or String | |
| Char | Character | Char | |
| boolean | Boolean | Boolean or String | |