```
Wrapper class
============
Purpose
 1. To wrap primitives into object form so that we can handle primitives
also just like objects.
 2. To define several utility functions which are required for the
primitives.

Constructors
===========
 Almost all the Wrapper class have 2 constructors
   a. one taking primitive type.
   b. one taking String type.

eg: Integer i=new Integer(10);
      Integer i=new Integer("10");

    Double d=new Double(10.5);
    Double d=new Double("10.5");

Note: If String argument is not properly defined then it would result in
RunTimeException called
      "NumberformatException".
      eg:: Integer i=new Integer("ten");//RE:NumberFormatException

Wrapper class and its associated constructor
   Byte   => byte and String
   Short  => short and String
  Integer => int and String
  Long    => long and String
  **Float => float ,String and double
   Double => double and String
 **Character=> character
 ***Boolean  => boolean and String

eg::
 1) Float f=new Float (10.5f);
 2) Float f=new Float ("10.5f");
 3) Float f=new Float(10.5);
 4) Float f=new Float ("10.5");

eg::
 1) Charcter c=new Character('a');
 2) Character c=new Character("a"); //invalid

eg::
Boolean b=new Boolean(true);
Boolean b=new Boolean(false);
Boolean b1=new Boolean(True);//C.E
Boolean b=new Boolean(False);//C.E
Boolean b=new Boolean(TRUE);//C.E

eg::
 Boolean b1=new Boolean("true");
 Boolean b2=new Boolean("True");
 Boolean b3=new Boolean("false");
 Boolean b4=new Boolean("False");
 Boolean b5=new Boolean("nitin");
 Boolean b6=new Boolean("TRUE");
 System.out.println(b1);//true
 System.out.println(b2);//true
 System.out.println(b3);//false
 System.out.println(b4);//false
 System.out.println(b5);//false
 System.out.println(b6);//true
```

eg::

```
Boolean b1=new Boolean("yes");
Boolean b2=new Boolean("no");
System.out.println(b1);//false
System.out.println(b2);//false
System.out.println(b1.equals(b2));//true
System.out.println(b1==b2);//false
```

eg::

```
Integer i2 = new Integer(10);
System.out.println(i1); //10
System.out.println(i1.equals(i2)); //true
```

Note: In case of Boolean constructor, boolean value be treated as true w.r.t to case insensitive
part of "true",for all others it would be treated as "false".

Note: If we are passing String argument then case is not important and content is not important.
If the content is case insensitive String of true then it is treated as true in all other cases it is treated as false.

Note: In case of Wrapper class,toString() is overriden to print the data.
In case of Wrapper class,equals() is overriden to check the content.
Just like String class, Wrapper classes are also treated as "Immutable class".

Can we create our own Immutable class

Yes, we can create our own Immutable class as shown below.

```
final public class Test  {
      private int i;

      Test(int i){
            this.i=i;
      }
      public Test modify(int i){
              if(this.i==i){
                        return this;
              }else{
                   return      new Test(i);
              }
      }
      public static void main(String[] args){
                  Test t=new Test(10);

                  Test t1= t.modify(10);
                  System.out.println(t==t1);//true
                  Test t2=t.modify(100);
                  System.out.println(t==t2);//false

                  Test t3=t2.modify(100);
                  System.out.println(t2==t3);//true

      }
}
```

Wrapper class utiltiy methods
=============================
1. valueOf() method.
2. XXXValue() method.
3. parseXxx() method.
4. toString() method.

valueOf() method
================
To create a wrapper object from primitive type or String we use valueOf().
It is alternative to constructor of Wrapper class, not suggestable to use.
Every Wrapper class,except character class contain static valueOf() to create a Wrapper Object.

eg#1.
```
Integer i=Integer.valueOf("10");
Double  d=Double.valueOf("10.5");
Boolean b=Boolean.valueOf("nitin");
   System.out.println(i);
   System.out.println(d);
   System.out.println(b);
```

eg#2.
```
     public static valueOf(String s,int radix)
                                  |=> binary : 2(0,1)
                                  |=> octal  : 8(0-7)
                                  |=> decimal : 10(0-9)
                                  |=> hexadecimal : 16(0-9,a,b,c,d,e,f)
                                  |=> base : 36(0-9,a-z)

Integer i1=Integer.valueOf("1111");
    System.out.println(i1);//1111
Integer i2=Integer.valueOf("1111",2);
    System.out.println(i2);//15
Integer i3=Integer.valueOf("ten");
    System.out.println(i3);//RE:NumberFormatException
Integer i4=Integer.valueOf("1111",37);
    System.out.println(i4);//RE:NumberFormatException
```

eg#3.
```
    public static valueOf(primitivetype x)

Integer i1=Integer.valueOf(10);
Double  d1=Double.valueOf(10.5);
Character c=Character.valueOf('a');
Boolean b=Boolean.valueOf(true);
   Primtive/String =>valueOf() => WrapperObject
```

```
2. xxxValue()
     We can use xxxValue() to get primitive type for the given Wrapper
Object.
     These methods are a part of every Number type Object.
     (Byte,Short,Integer,Long,Float,Double) all these classes have these
6 methods which is
      Written as shown below.

Methods
=======
  public byte byteValue();
  public short shortValue();
  public int intValue();
  public long longValue();
  public float floatValue();
  public double doubleValue();

eg#1.
 Integer i=new Integer(130);
 System.out.println(i.byteValue());//-126
 System.out.println(i.shortValue());//130
 System.out.println(i.intValue());//130
 System.out.println(i.longValue());//130
 System.out.println(i.floatValue());//130.0
 System.out.println(i.doubleValue());//130.0

3. charValue()
       Character class contains charValue() to get Char primitive for the
given Character
       Object.
      public char charValue()
eg#1.
  Character c=new Character('c');
  char ch= c.charValue();
  System.out.println(ch);

4. booleanValue()
         Boolean class contains booleanValue() to get boolean primitive
for the given boolean
         Object.
      public boolean booleanValue()

eg#1.
   Boolean b=new Boolean("nitin");
   boolean b1=b.booleanValue();
    System.out.println(b1);//false

In total xxxValue() are 36 in number.
 => xxxValue() => convert the Wrapper Object => primitive.


parseXXXX()
===========
 We use parseXXXX() to convert String object into primitive type.

form-1
======
public static primitive parseXXX(String s)
```

```
  Every wrapper class,except Character class has parseXXX() to convert
String into primitive type.

eg: int i=Integer.parseInt("10");
    double d =Double.parseInt("10.5");
    boolean b=Boolean.parseBoolean("true");


form-2
======
public static primitive parseXXXX(String s, int radix)
                                     |=> range is from 2 to 36

Every Integral type Wrapper class(Byte,Short,Integer,Long) contains the
following parseXXXX()
to convert Specified radix String to primitive type.

eg: int i=Integer.parseInt("1111",2);
    System.out.println(i);//15

Note: String => parseXXX() => primitive type

toString()
=========
 To convert the Wrapper Object or primitive to String.

Every Wrapper class contain toString()


form1
=====
 public String toString()

1. Every wrapper class (including Character class) contains the above
toString()
     method to convert wrapper object to String.
2. It is the overriding version of Object class toString() method.
3. Whenever we are trying to print wrapper object reference internally
this
   toString() method only executed

eg:  Integer i=Integer.valueOf("10");
     System.out.println(i);//internally it calls toString() and prints
the Data.

form2
=====
   public static String toString(primitivetype)

1. Every wrapper class contains a static toString() method to convert
primitive to String.

String s=Integer.toString(10);
                           |=> primitive type int.

eg:
 String s=Integer.toString(10);
 String s=Boolean.toString(true);
 String s=Character.toString('a');
```

```
form3
=====
Integer and Long classes contains the following static toString() method
to convert the
primitive to specified radix String form.


 public static String toString(primitive p,int radix)
                                   |=> 2 to 36

eg: String s=Integer.toString(15,2)
    System.out.println(s); // 1111


form4
=====
Integer and Long classes contains the following toXxxString() methods.
public static String toBinaryString(primitive p);
public static String toOctalString(primitive p);
public static String toHexString(primitive p);

Example:
class WrapperClassDemo {
   public static void main(String[] args) {
      String s1=Integer.toBinaryString(7);
      String s2=Integer.toOctalString(10);
      String s3=Integer.toHexString(20);
      String s4=Integer.toHexString(10);

      System.out.println(s1);//111
      System.out.println(s2);//12
      System.out.println(s3);//14
      System.out.println(s4);//a
  }
}

Snippets
=======
Examples :
Integer i1= new Integer(10);
Integer i2= new Integer(10);
System.out.println(i1==i2);//false

Integer i1 = 10;
Integer i2 = 20;
System.out.println(i1==i2);//true

Integer i1 =Integer.valueOf(10);
Integer i2 =Integer.valueOf(10);
System.out.println(i1==i2);//true

Integer i1 =10;
Integer i2 =Integer.valueOf(10);
System.out.println(i1==i2);//true

Note:
When compared with constructors it is recommended to use valueOf() method
to create wrapper object.
```

Case 1: Widening vs Autoboxing

Widening:
Converting a lower data type into a higher data type is called widening.

Example:

```
class AutoBoxingAndUnboxingDemo {
      public static void methodOne(long l) {
                System.out.println("widening");
      }
      public static void methodOne(Integer i) {
                System.out.println("autoboxing");
      }
      public static void main(String[] args) {
                int x=10;
                methodOne(x);
      }
}
```

Output:
Widening

Widening dominates Autoboxing.
Case 2: Widening vs var-arg method :

Example:

```
       import java.util.*;
            class AutoBoxingAndUnboxingDemo {
                public static void methodOne(long l) {
                        System.out.println("widening");
                }
                public static void methodOne(int... i) {
                        System.out.println("var-arg method");
                }
                public static void main(String[] args) {
                        int x=10;
                         methodOne(x);
                }
      }
```

Output:
Widening

case4:

```
class AutoBoxingAndUnboxingDemo {
      public static void methodOne(Integer i) {
                System.out.println("Autoboxing");
      }
      public static void methodOne(int... i) {
                System.out.println("var-arg method");
      }
      public static void main(String[] args) {
                int x=10;
                methodOne(x);
      }
}
```

Output:
Autoboxing

Case 5:

```
class AutoBoxingAndUnboxingDemo {
      public static void methodOne(Long l) {
            System.out.println("Long");
      }
      public static void main(String[] args) {
                  int x=10;
                  methodOne(x);
      }
}
```

Output:
```
methodOne(java.lang.Long) in AutoBoxingAndUnboxingDemo cannot be applied
to (int)
```

Note:
Widening followed by Autoboxing is not allowed in java but Autoboxing followed by widening is allowed.

Autoboxing dominates var-arg method.
In general var-arg method will get least priority i.e., if no other method matched then only var-arg method will get chance.
It is exactly same as "default" case inside a switch.

Note : While resolving overloaded methods compiler will always gives the presidence in the following order :
1. Widening
2. Autoboxing
3. Var-arg method.

Case 6:
```
class AutoBoxingAndUnboxingDemo {
            public static void methodOne(Object o) {
                  System.out.println("Object");
            }
            public static void main(String[] args) {
                  int x=10;
                   methodOne(x);
            }
}
```

Output:
```
Object
```

Which of the following declarations are valid ?
```
1. int i=10 ; //valid
2. Integer I=10 ; //valid
3. int i=10L ; //invalid CE :
4. Long l = 10L ; // valid
5. Long l = 10 ; // invalid CE :
6. long l = 10 ; //valid
7. Object o=10 ; //valid (autoboxing followed by widening)
8. double d=10 ; //valid
9. Double d=10 ; //invalid
10. Number n=10; //valid (autoboxing followed by widening)
```

```
Var arg method
==============
  It stands for variable argument methods.
  In java language,if we have variable no of arguments, then compulsorily
new method has to be
  written till jdk1.4.
  But jdk1.5 version, we can write single method which can handle
variable no of
  arguments(but all of them should be of same type).

  Syntax:: methodOne(dataType... varaibleName)
                ... => It stands for ellipse

eg1::
    class Demo
    {
     public void methodOne(int... x){System.out.println("var arg
method");}

        public static void main(String[] args){
                Demo d= new Demo();
            d.methodOne();//var arg method
            d.methodOne(10);//var arg method
            d.methodOne(10,20,30);// var arg method
      }
    }

Note:: internally the var arg method will converted to SingleDimension
Array, so we can access the
             var arg method arguments using index.

eg2::
  class Demo
    {
     public void methodOne(int... x){
            int total=0;
            for(int i=0;i<x.length;i++){
                  total+=x[i];
            }
            System.out.println("The sum is "+total);
      }
        public static void main(String[] args){
                Demo d= new Demo();
            d.methodOne();//The sum is 0
            d.methodOne(10);//The sum is 10
            d.methodOne(10,20,30);// The sum is 60
      }
    }

eg3::
  class Demo
    {
     public void methodOne(int... x){
            int total =0;
            for(int data:x){total+=data;}
                 System.out.println("The sum is "+total);
      }
        public static void main(String[] args){
                Demo d= new Demo();
            d.methodOne();//The sum is 0
            d.methodOne(10);//The sum is 10
            d.methodOne(10,20,30);// The sum is 60
      }
    }
```

Note::

case1
=====
Valid Signatures

```
 1.public void methodOne(int... x)
 2.public void methodOne(int...x)
 3.public void methodOne(int ...x)
```

case2
=====
We can mix normal argument with var argument

```
    public void methodOne(int x,int... y)
    public void methodOne(String s,int... x)
```

case3
=====
While mixing var arg with normal argument var arg should be always last.

```
 public void methodOne(int... x,int y); (invalid)
```

case4
=====
In an argument list there should be only one var argument

```
 public void methodOne(int... x,int ...y); (invalid)
```

case5
=====
We can overload var arg method, but var arg method will get a call only if none of matches are found.
(just like default statement of switch case)

eg::

```
 class Test
   {
      public void methodOne(int ...i){System.out.println("Var arg 
method");}
      public void methodOne(int i){System.out.println("Int arg method");}

      public static void main(String[] args)
      {
           Test t= new Test();
                  t.methodOne(10);//Int arg method
            t.methodOne();//Var arg method
            t.methodOne(10,20,30);//Int arg method
      }
   }
```

case6
=====

```
  public void methodOne(int... x) => it can be replace as int[] x
```

case7
=====

```
  public void methodOne(int... x)
  public void methodOne(int[] x)
  output:: CE because we cannot have two methods with same signature.
```

SingleDimension Array vs Var Arg method
=======================================

```
   1. Whereever Singledimesion array is present we can replace it with
Var Arg.
        eg:: public static void main(String[] args) => String... args
   2. Whereever Var arg  is present we cannot replace it with
SingleDimension Array.
        eg:: public void methodOne(String... args) => String[] args
(invalid)
```

Note::
eg1::

```
 class Test
 {
        public void methodOne(int... x){
           for(int data: x){
                System.out.println(data);
           }
         }
      public static void main(String... args){
                    Test t= new Test();
                    t.methodOne(10,20,30);
        }
 }
```

In the above pgm x is treated as One-D array.

eg2::

```
class Test
 {
        public void methodOne(int[]... x){
           for(int[] OneD: x){
           for(int element:oneD){
             System.out.println(data);
                }
           }
         }
      public static void main(String... args){
                    Test t= new Test();
               int[] a= {10,20,30};
                    int[] b= {30,40};

                    t.methodOne(a,b);
        }
 }
```

In the above program x is treated as 2D array

```
Note:: methodOne(int...x)
            => we can call this method by passing a group of int
values,so it becomes 1-D array.
       methodOne(int[]... x)
            => we can call this method by passing a group of 1D int[], so
it becomes 2-D array.
```

Import statement
==============
```
class Test{
      public static void main(String args[]){
            ArrayList l=new ArrayList();
      }
}
```
Output:
Compile time error.
```
D:\Java>javac Test.java
Test.java:3: cannot find symbol
symbol : class ArrayList
location: class Test
ArrayList l=new ArrayList();
```

=> We can resolve this problem by using fully qualified name "java.util.ArrayList"
      l=new java.util.ArrayList();". But problem with using fully qualified name every time is it increases length of the code and
      reduces readability.
=> We can resolve this problem by using import statements.

Example:
```
import java.util.ArrayList;
class Test{
      public static void main(String args[]){
                  ArrayList l=new ArrayList();
      }
}
```
Output:
```
D:\Java>javac Test.java
```
Hence whenever we are using import statement it is not require to use fully qualified names we can use short names directly.
This approach decreases length of the code and improves readability.

Case 1: Types of Import Statements:
There are 2 types of import statements.
      1) Explicit class import
      2) Implicit class import.

Explicit class import:
Example: Import java.util.ArrayList ;
      => This type of import is highly recommended to use because it improves readability of the code.
      => Best suitable for developers where readability is important.

Implicit class import:
Example: import java.util.*;
=> It is never recommended to use because it reduces readability of the code.
=> Best suitable for students where typing is important.

Case 2:
Which of the following import statements are meaningful ?
```
      import java.util;
      import java.util.ArrayList.*;
      import java.util.*;
      import java.util.ArrayList;
```

Case3:
consider the following code.

```
class MyArrayList extends java.util.ArrayList
{

}
```

=> The code compiles fine even though we are not using import statements because we used fully qualified name.
=> Whenever we are using fully qualified name it is not required to use import statement.
Similarly whenever we are using import statements it is not require to use fully qualified name.

Case4:

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

Output:
Compile time error.

```
D:\Java>javac Test.java
Test.java:7: reference to Date is ambiguous,
both class java.sql.Date in java.sql and class java.util.Date in
java.util match

Date d=new Date();
```

Note: Even in the List case also we may get the same ambiguity problem because it is available in both util and awt packages.

Case5:
While resolving class names compiler will always gives the importance in the following order.

1. Explicit class import
2. Classes present in current working directory.
3. Implicit class import.

Example:

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

The code compiles fine and in this case util package Date will be considered.

Case 6:
Whenever we are importing a package all classes and interfaces present in that package are by default available but not
sub package classes.

```
      java
        |=> util
              |=> regex
                    |=> Pattern
```

To use pattern class in our Program directly which import statement is required ?

```
      import java.*;
      import java.util.*;
      import java.util.regex.*;
      import java.util.regex.Pattern;
```

Case7:
In any java Program the following 2 packages are not require to import because these are available by default to every java Program.
1. java.lang package
2. default package(current working directory)

Case 8:
"Import statement is totally compile time concept" if more no of imports are there then more will be the compile time
 but there is "no change in execution time".

Difference between C language #include and java language import ?
#include
=======
1. It can be used in C & C++
2. At compile time only compiler copy the code from standard library and placed in current program.
3. It is static inclusion
4. wastage of memory
Ex : <jsp:@ file="">

import
======
1. It can be used in Java
2. At runtime JVM will execute the corresponding standard library and use it's result in current program.
3. It is dynamic inclusion
4. No wastage of memory
 Ex : <jsp:include >

Note:
 In the case of C language #include all the header files will be loaded at the time of include statement hence it follows static loading.
 But in java import statement no ".class" will be loaded at the time of import statements in the next lines of the code whenever we are
 using a particular class then only corresponding ".class" file will be loaded. Hence it follows "dynamic loading" or
 "load-on -demand" or "load-on-fly".

1.5 versions new features :
1. For-Each
2. Var-arg
3. Queue
4. Generics
5. Auto boxing and Auto unboxing
6. Co-varient return types
7. Annotations
8. Enum

9.   Static import
10. String builder

Static import:
This concept introduced in 1.5 versions. According to sun static import improves readability of the code but according to worldwide Programming exports (like us) static imports creates confusion and reduces readability of the code. Hence if there is no specific requirement never recommended to use a static import.

Usually we can access static members by using class name but whenever we are using static import it is not require to use class name we can access directly.

Without static import:
```
class Test
{
      public static void main(String args[]){
                  System.out.println(Math.sqrt(4));
                  System.out.println(Math.max(10,20));
                  System.out.println(Math.random());
      }
}
```
Output:
```
D:\Java>javac Test.java
D:\Java>java Test
2.0
20
0.841306154315576
```

With static import:
```
import static java.lang.Math.sqrt;
import static java.lang.Math.*;

class Test{
      public static void main(String args[]){
                  System.out.println(sqrt(4));
                  System.out.println(max(10,20));
                  System.out.println(random());
      }
}
```
Output:
```
D:\Java>javac Test.java
D:\Java>java Test
2.0
20
0.4302853847363891
```

Example 3:
```
import static java.lang.System.out;
class Test{
      public static void main(String args[]){
            out.println("hello");
            out.println("hi");
      }
}
```
Output:
```
D:\Java>javac Test.java
D:\Java>java Test
```

```
hello
hi
```

Example 4:

```
import static java.lang.Integer.*;
import static java.lang.Byte.*;
class Test{
      public static void main(String args[]){
                  System.out.println(MAX_VALUE);
      }
}
```

Output:
Compile time error.

```
D:\Java>javac Test.java
Test.java:6: reference to MAX_VALUE is ambiguous,
both variable MAX_VALUE in java.lang.Integer and variable MAX_VALUE in
java.lang.Byte match
      System.out.println(MAX_VALUE);
```

Note:
Two packages contain a class or interface with the same is very rare
hence ambiguity problem is very rare in normal import.
But 2 classes or interfaces can contain a method or variable with the
same name is very common hence ambiguity
problem is also very common in static import.

While resolving static members compiler will give the precedence in the
following order.
1. Current class static member
2. Explicit static import
3. implict static import

eg:

```
import static java.lang.Integer.MAX_VALUE;
import static java.lang.Byte.*;
class Test{
      static int MAX_VALUE  = 999;
      public static void main(String[] args){
            System.out.println(MAX_VALUE);
      }
}
```

Which of the following import statement is valid?

```
 import java.lang.Math.*;
 import static java.lang.Math.*;
 import java.lang.Math;
 import static java.lang.Math;
 import static java.lang.Math.sqrt.*;
 import java.lang.Math.sqrt;
 import static java.lang.Math.sqrt();
 import static java.lang.Math.sqrt;
```

Usage of static import reduces readability and creates confusion hence if
there is no specific requirement never recommended to use static import.

What is the difference between general import and static import ?

normal import
============

 => We can use normal imports to import classes and interfaces of a package.
 => whenever we are using normal import we can access class and interfaces directly by their short name it is not
 require to use fully qualified names.

static import
===========
=> We can use static import to import static members of a particular class.
=> whenever we are using static import it is not require to use class name we can access static members directly.

Package statement:
It is an encapsulation mechanism to group related classes and interfaces into a single module.

The main objectives of packages are:
 => To resolve name confects.
 => To improve modularity of the application.
 => To provide security.
 => There is one universally accepted naming conversion for packages that is to use internet domain name in reverse.

eg: com.icicibank.loan.housingloan.Account

com.icicibank => client internet domain in reverse
loan=> module name
housingloan=> submodule
Account => Class

How to compile package Program:
```
package in.ineuron.main
public class Test{
      public static void main(String[] args){
            System.out.println("package demo");
      }
}
```
```
  javac  Test.java      => Test.class file will be generated in current working directory
  javac -d . Test.java => Test.class file will be generated inside in.ineuron.main.Test.class
```

Note:
-d means destination to place generated class files "." means current working directory.
Generated class file will be placed into corresponding package structure.

javac -d . Test.java
 If the specified package structure is not already available then this command itself will create the required package structure.
 As the destination we can use any valid directory.
 If the specified destination is not available then we will get compile time error

javac -d c: Test.java
 Test.class file will be created in C:\in\ineuron\main\Test.class

```
javac -d z: HydJobs.java
```
If Z: is not available then we will get compile time error.

How to execute package Program:

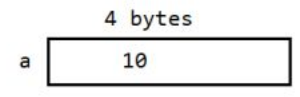
```
D:\Java>java in.ineuron.main.Test
```
At the time of execution compulsory we should provide fully qualified name.

Conclusion 1:
In any java Program there should be at most one package statement that is if we are taking more than one package statement
we will get compile time error.

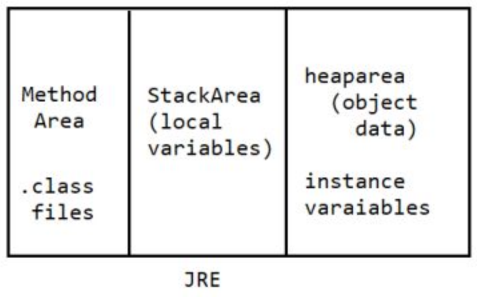
Example:
```
package pack1;
package pack2;
class A
{
}
```
Output:
Compile time error.
```
D:\Java>javac A.java
A.java:2: class, interface, or enum expected package pack2;
```

Conclusion 2:
In any java Program the 1st non comment statement should be package statement [if it is available] otherwise we will get compile time error.
Example:
```
import java.util.*;
package pack1;
class A
{
}
```
Output:
Compile time error.
```
D:\Java>javac A.java
A.java:2: class, interface, or enum expected package pack1;
```

```
int a = 10;
⇧
primitive type
```

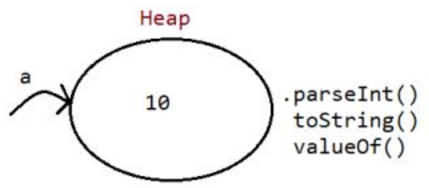


```
local variable    => stack
instance variable => heap
static variable   => methodarea(heaparea)
```

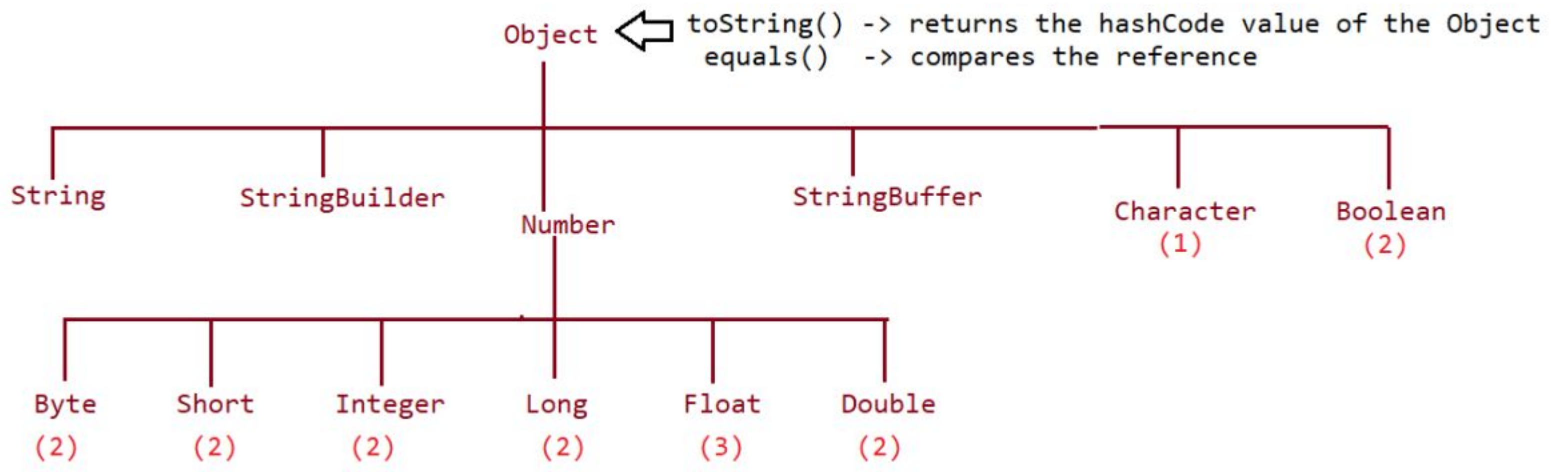


JDK1.5V Wrapper classes are introduced

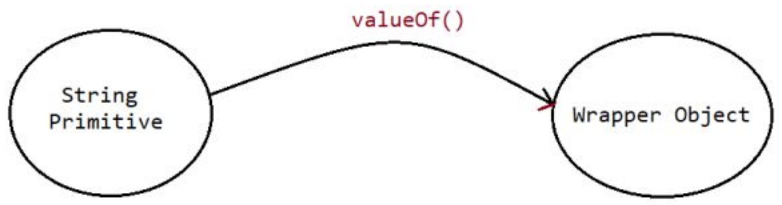
```
Integer a = 10;
⇧
reference type
```



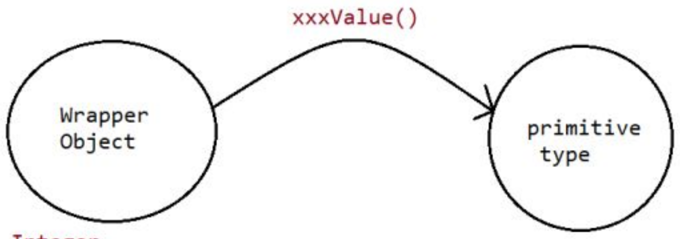
Present in "java.lang" package



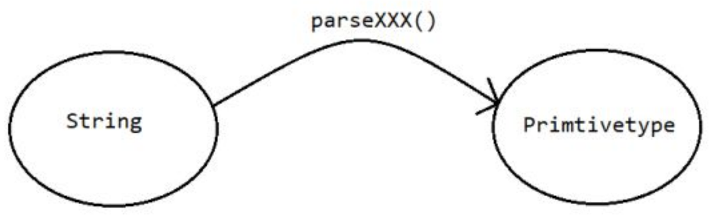
toString() => Overriden to print the data present in the Object
equals() => Overriden to compare the content present in the Object

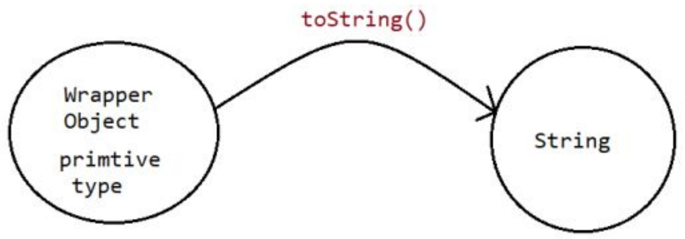


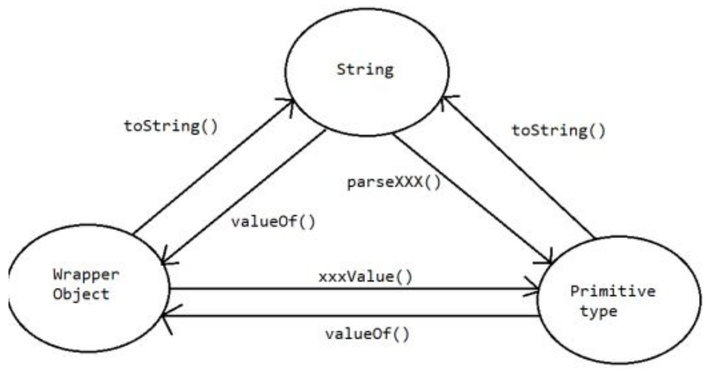
```
public static Character valueOf(char);
```



```
Integer
Byte
Short     =>6 xxxValue()
Long
Float
Double

booleanValue()
charValue()
```

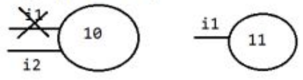




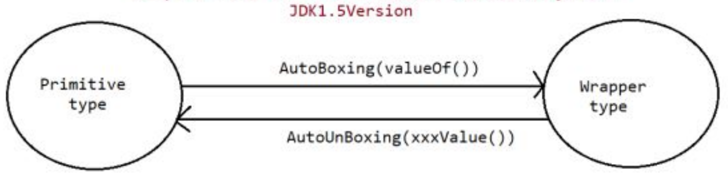


```
Integer i1 = 10;
Integer i2 = i1;
i1++;
System.out.println(i1); //11
System.out.println(i2); //10
System.out.println(i1==i2);//false
```

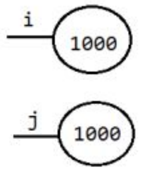
Integer(Immutable)



Compiler will do the conversions automtically from JDK1.5Version



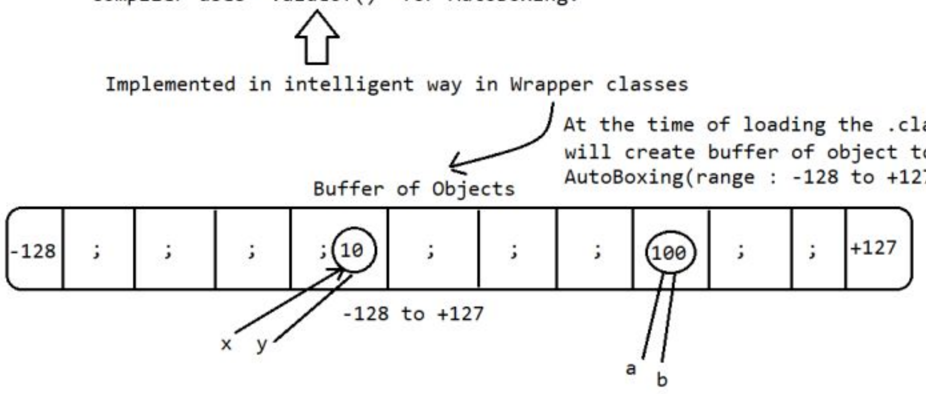
```
Integer x = 10;
Integer y = 10;
System.out.println(x == y);//true

Integer a = 100;
Integer b = 100;
System.out.println(a == b);//true

Integer i = 1000;
Integer j = 1000;
System.out.println(i == j);//false
```



Compiler uses "valueOf()" for AutoBoxing.

Implemented in intelligent way in Wrapper classes

At the time of loading the .class file jvm will create buffer of object to be used during AutoBoxing(range : -128 to +127)


Buffer of Objects
-128
;
;
;
;
10
;
;
;
100
;
;
+127
-128 to +127
x
y
a
b