Wrapper Classes (Main part - 5)

JDK8-features

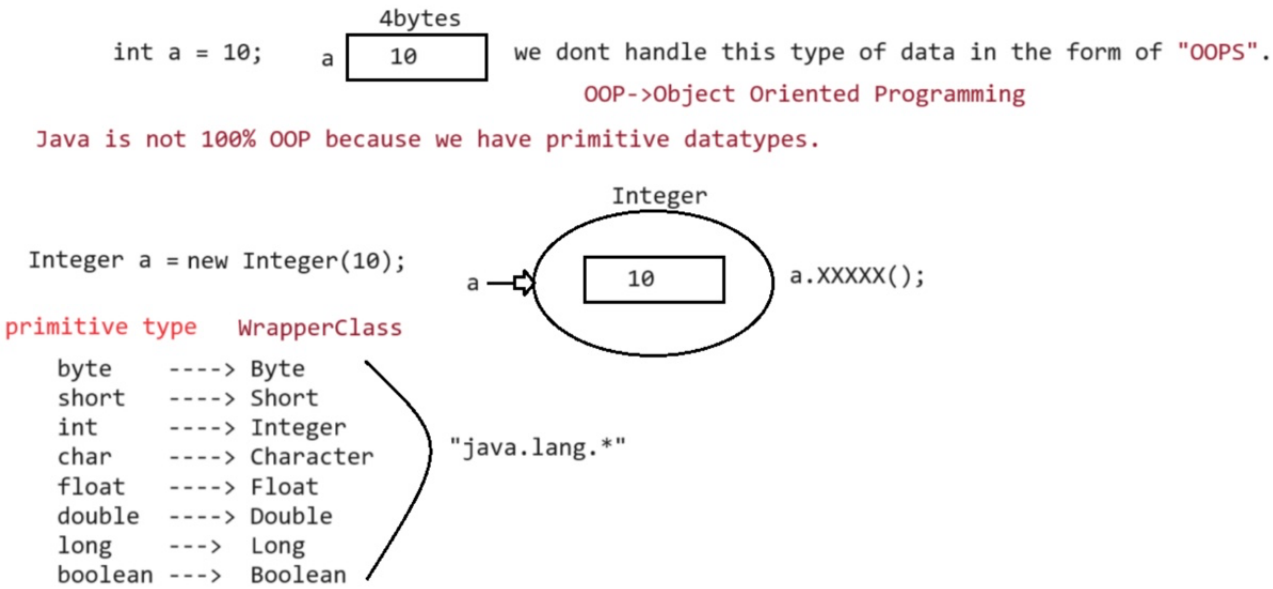
1. Functional interface
2. Lambda Expression,Method reference, Constructor reference
3. default and static methods in interface
4. Inbuilt functional interface
   a. Runnable
   b. Predicate
   c. Function
   d. Supplier
   e. Consumer
5. Optional API
6. StringJoiner
7. JodaAPI

Wrapper classes

What is the need of Wrapper class, when we already had primitive types like int,float,double,....

Ans. To wrap primitives into Object form, so that we can handle primitives also just like Objects. To define several utility methods which are requried for primitives.

```
int a= 10;//primtive data
System.out.println(a);
System.out.println(a.toString());//CE
```



Integer a = 10

or

Integer a = new Integer()

Both will make a object and store 10 inside of it. Now we can use '.' to access multiple methods available inside that wrapper class

Simple idea behind Wrapper class is to make java 100% OOP language.

Constructors

just type → javap java.lang.Integer → to check the constructors for Integer and other types

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**

Note: In wrapper class, toString() is overriden to print the content of the object which is similar to "String" class.

@Override

public java.lang.String toString(){

    //override to print the data present in the object

}

Integer i1 = new Integer(10);

So writing i1 or i1.toString() is same as internally toString() is applied on it.

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 → as there is only on character(no string)

***Boolean => boolean and String → no exception is thrown in constructor when string is passed so it will accept any string(even case sensitivity will not come in picture)

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"); **// all other strings will be result in false**

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 b7 = new Boolean("yes");

Boolean b8 = new Boolean("no");

System.out.println(b7); //false

System.out.println(b8); //false

System.out.println(b7 == b8); **//reference comparison :: false**

System.out.println(b7.equals(b8)); **//content comparison :: true**

eg::

Integer i2 = new Integer(10);

System.out.println(i1);

System.out.println(i1.equals(i2));

Output

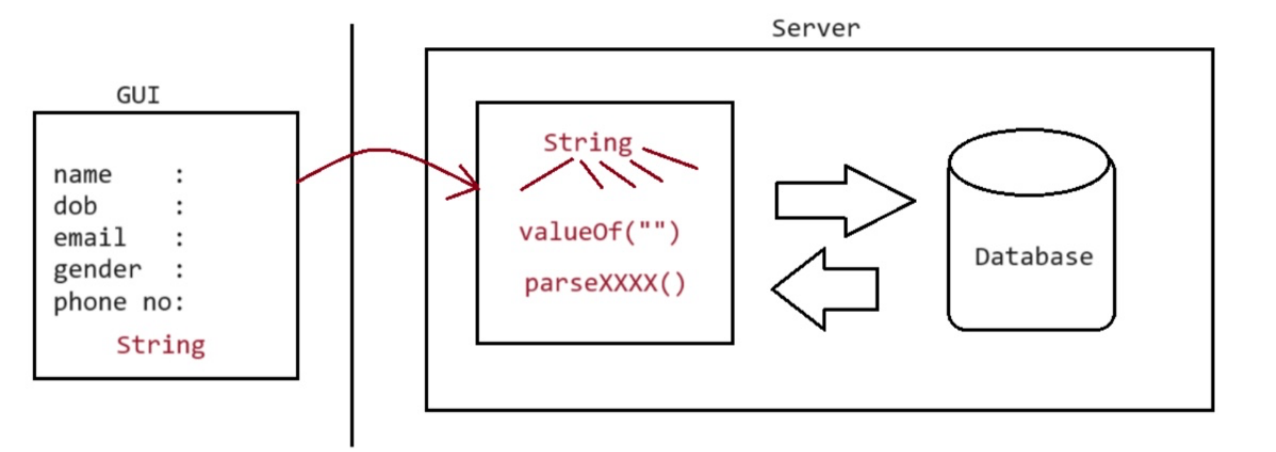
10

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".**



From frontend everything will come in the form of string. Before storing it in db we need to convert it into its respective which can easily acheived through Wrapper classes's valueOf or parseXXXX(where XXXX is a type) method.

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

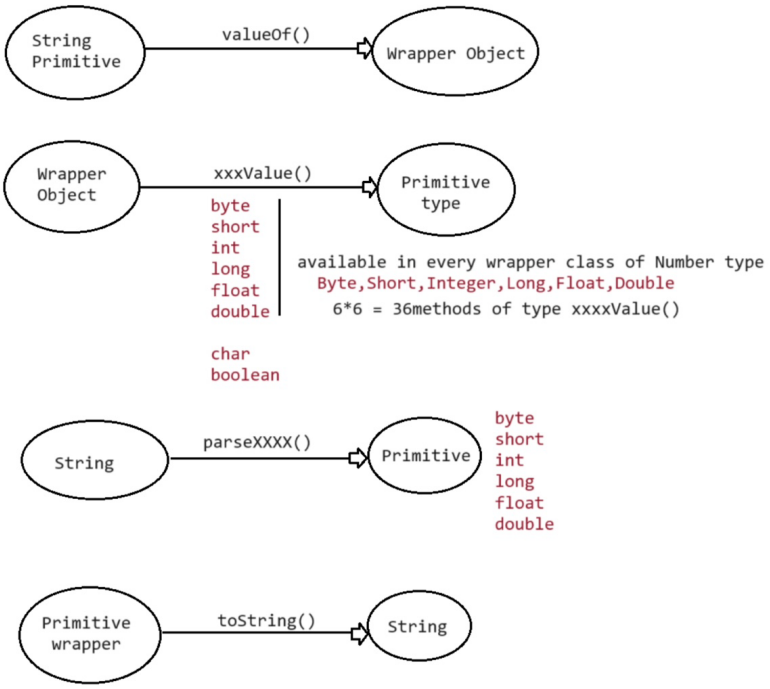
System.out.println(i.shortValue());//130

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



last one in above diagram → primitive or wrapper to string

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 c onvert 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.

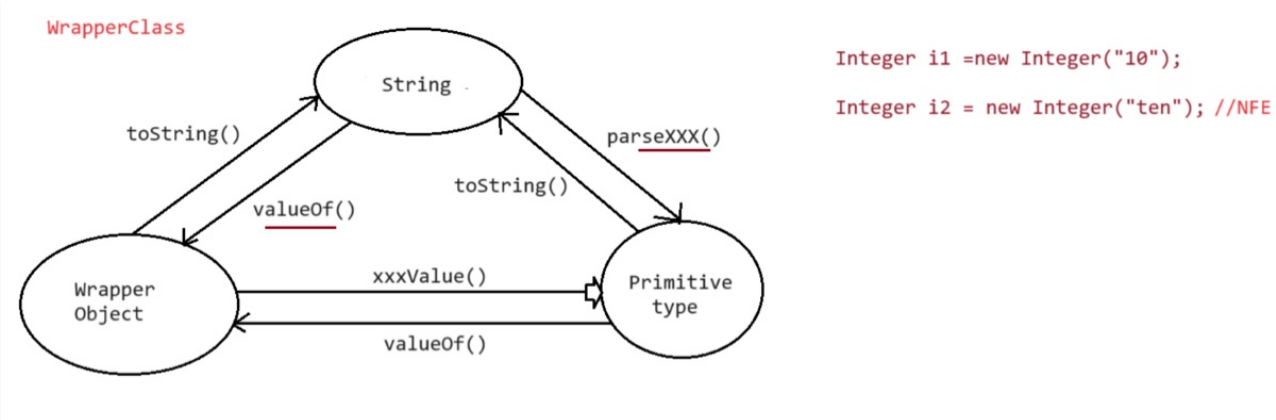
public static String toBinaryString(primitive p);

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
    }
}
```



1. **valueOf() :: static method**

   Primitive -> Wrapper

   String -> Wrapper

2. **toString() :: static methods**

   Wrapper -> String

   Primitive -> String

3. **xxxxValue() :: static methods**

   Wrapper -> Primitive

4. **parseXXXX() :: static method**

   String -> Primitive

Commonly used methods are "valueOf(),parseXXXX()"

Note:

1. String,StringBuffer,StringBuilder,Wrapper classes ===> final classes
2. Similar to String objects, wrapper classes object are also immutable.
3. Wrapper classes which are not direct child classes of object are

   Byte,Short,Integer,Long,Float,Double

Can we create our own class[UserDefined]as Immutable?

immutable means → if we make a change in an object than in existing object the changes will not reflect instead a new object will be created

Ans. yes ,it is possible to make userdefined class a Immutable class.

eg#1.

```
//UserDefined class
final class ImmutableClass
{
    private int i;
    ImmutableClass(int i){
        this.i= i;
    }
    public ImmutableClass modifyValue(int i){
        //if same value in existing object, dont' create new object
        if (this.i == i )
        {
            //share same reference
            return this;
        }
```

```
        //otherwise create new one
        else
        {
            return new ImmutableClass(i);
        }
    }
}
public class Test
{
    public static void main(String[] args)
    {
        ImmutableClass c1 = new ImmutableClass(10);
        ImmutableClass c2 = c1.modifyValue(10);
        ImmutableClass c3 = c1.modifyValue(20);
        System.out.println(c1==c2);//true
        System.out.println(c1==c3);//false
        System.out.println(c2==c3);//false
        ImmutableClass c4 = c1.modifyValue(10);
        System.out.println(c2==c4);//true
    }
}
```

Output

true

false

false

true

final vs Immutable vs Mutable

final :: variable[CompileTimeConstant],class[inheritance is not possible]

Immutable :: Objects[if we try to make a change to the object data, with that change new object will be created]

Mutable :: Objects[if we try to make a change to the object data changes will happen on the same data]

eg#1.

```
final StringBuffer sb = new StringBuffer("sachin");
sb.append("tendulkar")
```

System.out.pirntln(sb);//sachintendulkar

sb = new StringBuffer("dhoni"); **//CE: can't be reassigned**

AutoBoxing

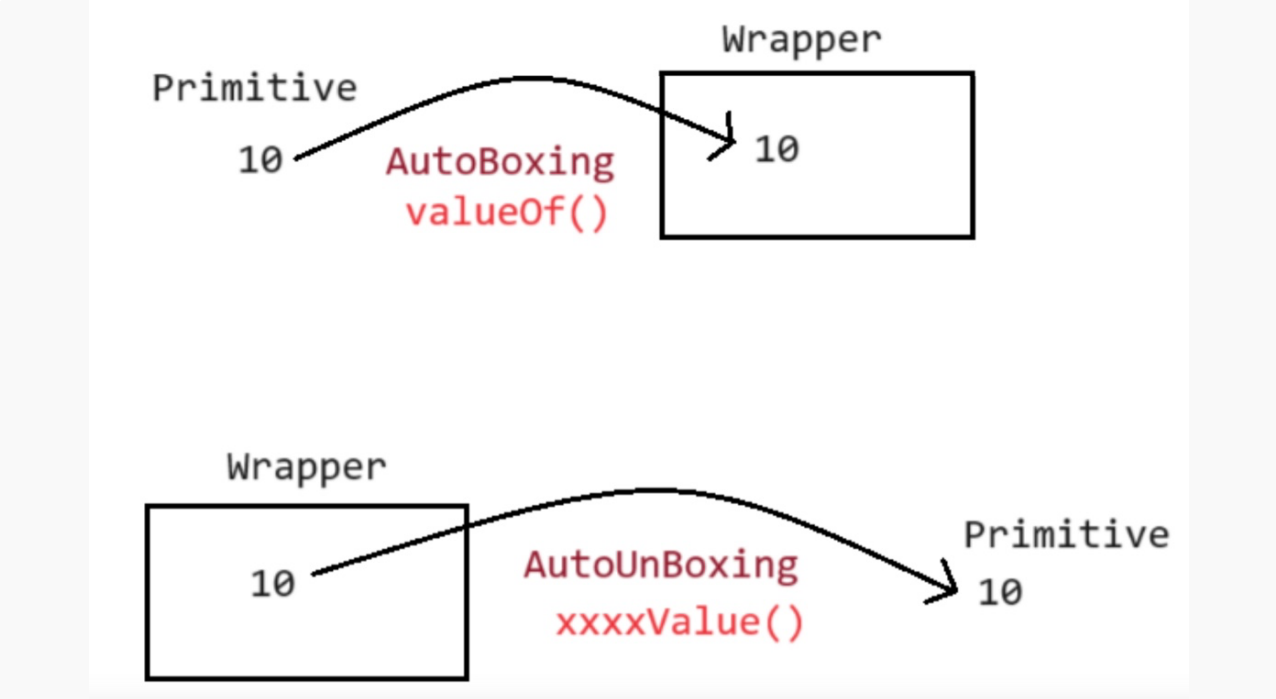
=> Automatic conversion of primitive to wrapper object by the compiler is called "AutoBoxing".

=> Internally compiler uses valueOf() to do "AutoBoxing".**(putting data in a box(wrapper))(auto - compiler is automatically doing it)**

AutoUnBoxing

=> Automatic conversion of wrapper to primtive data by the compiler is called "AutoUnBoxing".

=> Internally compiler uses xxxxValue() to do "AutoUnBoxing".**(removing data from box)(auto - compiler is automatically doing it)**



eg#1.

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.println("**AutoBoxing**");
        int i3 = 10;
        //Integer i4 = Integer.valueOf(i3); :: AutoBoxing
        Integer i4 = i3;//Primitive --->Wrapper
        System.out.println(i3);
```

```
        System.out.println(i4);
        System.out.println();
        Integer i1 = new Integer(10);
        //int i2 = i1.intValue(); :: AutoUnBoxing
        int i2 = i1;// Wrapper ---> Primitive
        System.out.println("AutoUnBoxing**");
        System.out.println(i1);
        System.out.println(i2);
    }
}
```

Output

AutoBoxing**

10

10

AutoUnBoxing**

10

10

eg#2.

```
public class Test
{
    static Integer i1 = 10; //AutoBoxing :: valueOf()
    public static void main(String[] args)
    {
        int i2 = i1;   //AutoUnBoxing :: intValue()
        methodOne(i2);
    }
    //AutoBoxing :: valueOf()
    public static void methodOne(Integer i3){
        int k = i3;  //AutoUnBoxing :: intValue()
        System.out.println(k);
    }
}
```

Output

10

eg#3.

```
public class Test
{
    static Integer i1 = 0;//AB::valueOf()
    public static void main(String[] args)
    {
        int i2 = i1;//AUB: intValue()
        System.out.println(i2);
    }
}
```

Output

0

eg#4.

```
public class Test
{
    static Integer i1 = null;//AB::valueOf()
    public static void main(String[] args)
    {
        int i2 = i1;   //AUB: Integer.intValue() :: NPE
        System.out.println(i2);
    }
}
```

Since i1 is a refrence type so it can hold refrence type but i2 is a primitive type which can just hold values so on conversion it will throw **NPE.**

Imp example → Wrapper classes are immutable

eg#5.

```
public class Test
{
    public static void main(String[] args)
    {
        Integer x = 10;
        Integer y = x;
        ++x;
        System.out.println(x);
        System.out.println(y);
        System.out.println(x==y);
```

```
    }
}
```

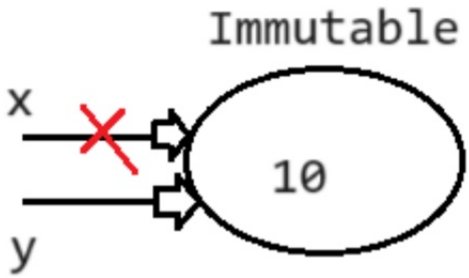
Output

11

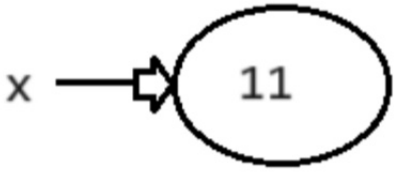
10

false

```
Integer x = 10;
Integer y = x;
```




```
++x;
```




```
System.out.println(x);
System.out.println(y);
System.out.println(x==y);
```

eg#1.

Integer i1= new Integer(10);

Integer i2= new Integer(10);

System.out.println(i1==i2); **//false**

eg#2.

Integer x = new Integer(10);

Integer y = 10;

System.out.println(x==y); **//false**

eg#3.

Integer x = new Integer(10);

Integer y = x;

System.out.println(x==y); **//true**

eg#4.

Integer i1 = 10;

Integer i2 = 10;

System.out.println(i1==i2); **//true**

eg#5.

Integer i1 = 100;

Integer i2 = 100;

System.out.println(i1==i2); **//true**

eg#6.

Integer i1 = 1000;

Integer i2 = 1000;

System.out.println(i1==i2); **//false**

Note:

1. To implement AutoBoxing concept in every wrapper class a buffer of objects will be created at the time of loading the .class file.
2. By AutoBoxing, if an object is requried to create 1st JVM will check whether that object is avaiable in buffer or not.
3. If it is avaiable then JVM wil reuse the buffer object instead of creating a new object.
4. If the object is not available in the buffer then try to create a new object.
5. By doing so memory will be effectively utilized and it improves the application performance.
6. **Performance is improved because buffer object is already created and available to use and we need not to make a new object at that time. For this reason new keyword is deprecated in newer versions. They have restricted the use of constructors for wrapper classes.**

Buffer concepts for wrapper class

a. Byte,Short,Integer,Long -> -128 + 127

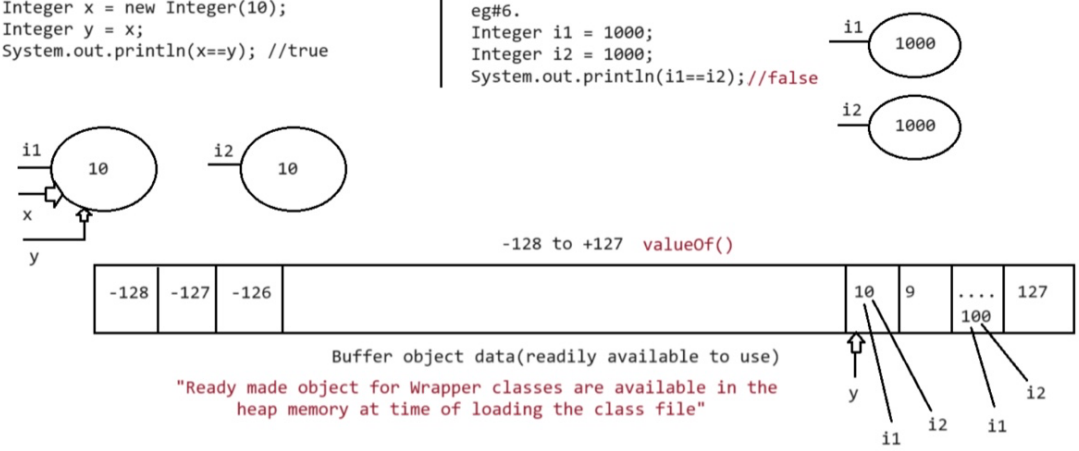
b. Character -> 0 to 127

c. Boolean -> true,false

In remaing cases compulsory new objects should be created.

```
eg#1.
Integer i1= new Integer(10);
Integer i2= new Integer(10);
System.out.println(i1==i2);  //false

eg#2.
Integer x = new Integer(10);
Integer y = 10;
System.out.println(x==y);  //false

eg#3.
Integer x = new Integer(10);
Integer y = x;
System.out.println(x==y); //true
```

```
eg#4.
Integer i1 = 10;
Integer i2 = 10;
System.out.println(i1==i2);//true

eg#5.
Integer i1 = 100;
Integer i2 = 100;
System.out.println(i1==i2);//true

eg#6.
Integer i1 = 1000;
Integer i2 = 1000;
System.out.println(i1==i2);//false
```



Since range is from -128 to 127 new objects will be created for 1000.

Snippets

Examples :

Integer i1= new Integer(10);

Integer i2= new Integer(10);

System.out.println(i1==i2); **//false**

Integer i1 = 10;

Integer i2 = 10;

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. In higher version of java9 and above, Creation of Wrapper class object using new keyword is "deprecated".

```
1.
Integer x = 127;
Integer y = 127;
System.out.println(x==y); //true

2.
Integer x = 128;
Integer y = 128;
System.out.println(x==y); //false

3.
Boolean b1 = true;
Boolean b2 = false;
System.out.println(b1==b2); //false
```

```
4.
Double d1 = 10.0;
Double d2 = 10.0;
System.out.println(d1==d2); //false

"No Buffer memory for Double type, always create a new object"
```

Boolean b1 = true;

Boolean b2 = true;

System.out.println(b1==b2); **//true**

Overloading w.r.t widening, autoboxing and var-arg method

Case 1: widening vs AutoBoxing

=> Widening always dominates AutoBoxing

```
public class Test
{
    //Widening method
    public static void methodOne(long l){
        System.out.println("long Version");
    }
    //AutoBoxing method
    public static void methodOne(Integer i){
        System.out.println("Integer Version");
    }
    public static void main(String[] args)
    {
        int x= 10;
        methodOne(x);
    }
}
```

Output

long version

Case2: Widening vs Var-args

=> Widnening dominates Var-args

```
public class Test
{
    //Widening method
    public static void methodOne(long l){
        System.out.println("widening...");
    }
    //AutoBoxing method
    public static void methodOne(int... i){
        System.out.println("Var-arg...");
    }
    public static void main(String[] args)
    {
        int x= 10;
        methodOne(x);
    }
}
```

Output

widening

Case3: AutoBoxing vs Var-args

-> AutoBoxing dominates Var-args

```
public class Test
{
    //Widening method
    public static void methodOne(Integer i){
        System.out.println("AutoBoxing... ");
    }
    //AutoBoxing method
    public static void methodOne(int... i){
        System.out.println("Var-arg...");
    }
    public static void main(String[] args)
    {
        int x= 10;
        methodOne(x);
    }
}
```

Output

AutoBoxing...

In general var-arg will get last priority,if no other method is matched then only var-arg method will get a chance. **It is exactly same as "default" case of switch statement.**

Note: While resolving the overloaded method compiler will always gives the precedence in the following order

1. widening
2. autoboxing
3. var-args

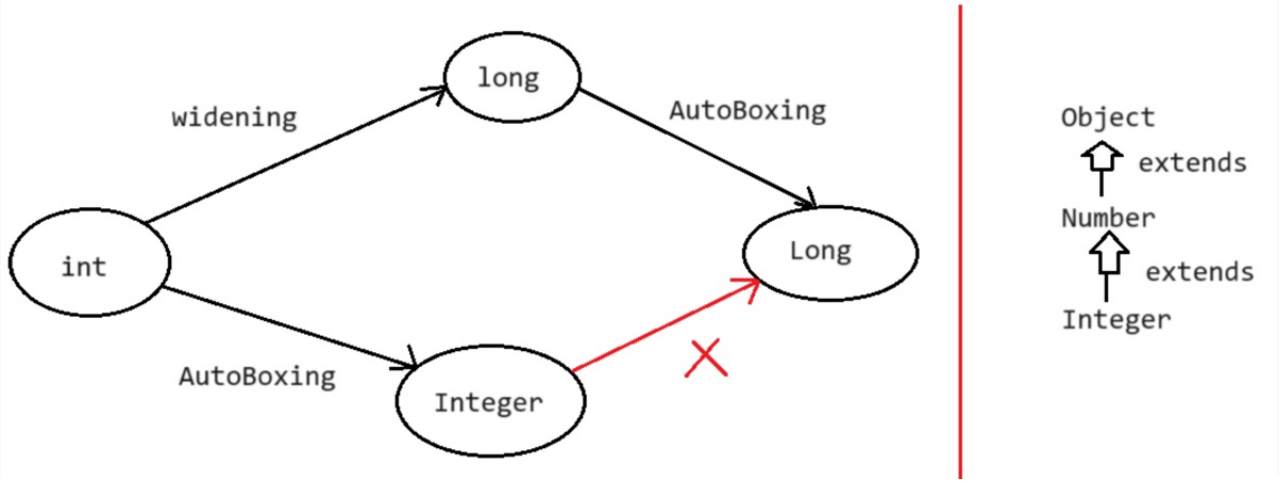
Case4:

```
public class Test
{
    //AutoBoxing method
    public static void methodOne(Long i){
        System.out.println("Long version");
    }
    public static void main(String[] args)
    {
        int x= 10;
        methodOne(x);//int---> Integer--notpossible--> Long
    }
}
```

Output

error: incompatible types: int cannot be converted to Long

Note: Widening followed by Autoboxing is allowed in java, where as AutoBoxing followed by Widening is not allowed in java.



So in conversion from int to Long, first widening and then autoboxing is possible but wise versa is not possible

case5:

```
public class Test
{
    //AutoBoxing method
    public static void methodOne(Object i){
        System.out.println("Object version");
    }
    public static void main(String[] args)
    {
        int x= 10;
        methodOne(x);//int---> Integer---->Object
    }
}
```

Output

Object version

In above example Integer is not present → parent - Integer is also not present → parent - Object is present. Hence possible.

Case6.

```
public class Test
{
    //AutoBoxing method
    public static void methodOne(Object i){
        System.out.println("Object version");
    }
    public static void methodOne(Number n){
        System.out.println("Number version");
    }
    public static void methodOne(Long l){
        System.out.println("Long version");
    }
    public static void methodOne(Byte b){
        System.out.println("Byte version");
    }
    public static void methodOne(byte b){
        System.out.println("byte version");
    }
```

```
    public static void methodOne(int... x){
        System.out.println("var-arg version");
    }
    public static void main(String[] args)
    {
        int x= 10;
        methodOne(x);//int ----> Integer,Number,Object
    }
}
```

Output

Number version

Question

Which of the following declarations are valid ?

1. int i=10 ; **//valid**
2. Integer I=10 ; **//AutoBoxing :: valueOf()**
3. int i=10L ; **//invalid**
4. Long l = 10L ; **//AutoBoxing :: valueOf()**
5. Long l = 10 ; **//Invalid**
6. long l = 10 ; **//valid**
7. Object o=10 ; **//int----> Integer ---> Number---> Object**
8. double d=10 ; **//valid**
9. Double d=10 ; **//Invalid**
10. Number n=10; **//int---> Integer ---> Number**

Left out points →

++

not done -->

1)- 22 —> snippets are not done as it includes & , |

2)- **ALL THE INHERITANCE SNIPPETS ARE NOT SOLVED**

3)-

Parent p1 = new Child();

System.out.println(p1.x); **//888**

4)- **LEFT POINTS —> IMPLICIT AND EXPLICIT TYPECASTING, 29 and forward no images added and from 3rd onward no notes are also not added**

after the above line all the snippets are not solved.

5)- **30th dec first 40 mins questions are not solved**

++

not clear →

Q>

int[][] a =new int[3][2];

a[0] = new int[3];

a[1] = new int[4];

a = new int[4][3];

Totally how many objects are created?

Ans. 11

How many objects are eligible for Garbage Collection?

Ans. 6

Q)-

```
public class Boxer1{
    Integer i;
    int x;
    public Boxer1(int y) {
        x = i+y;   // x = null + 4
        System.out.println(x);
    }
    public static void main(String[] args) {
        new Boxer1(new Integer(4));
    }
}
```

What is the result?

A. The value "4" is printed at the command line.

B. Compilation fails because of an error in line 5.

C. Compilation fails because of an error in line 9.

D. A NullPointerException occurs at runtime.

E. A NumberFormatException occurs at runtime.

F. An IllegalStateException occurs at runtime.

Answer: D

3)- How is this a runtime polymorphism(Method-Overriding)(Method-Overriding :: LooseCoupling) ??

RunTime Polymphism[1:M]

public void allowAnimal**(Animal ref)**

{

ref.eat();

ref.sleep();

ref.breathe();

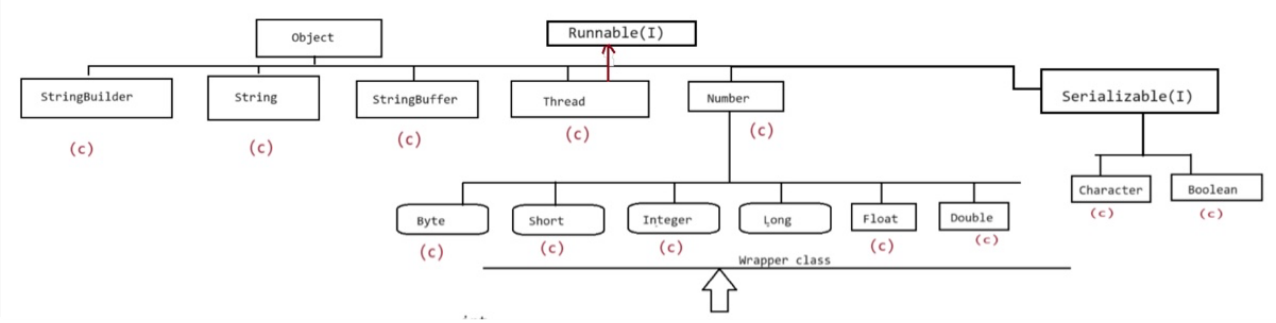
System.out.println();

}

Ans for 3 -The key to achieving true polymorphism here is that at compile time, Java only knows that you have a reference variable of type `Shape`. However, when you call `calculateArea()`, the actual method invoked depends on the object's type at runtime (either `Square` or `Circle` object). This allows for flexible and dynamic behavior based on the actual object instance.

++

++



Can Runnable only implemented by Thread??? → check once (imp)

++