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?
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".

AutoUnBoxing

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

=> Internally compiler uses xxxxValue() to do "AutoUnBoxing".

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);
      }
}
```

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
```

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.

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.

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".

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.

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
```

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
```

Collections
==========

int x=10; int y=20; int z=30;

In this approach, if i want to keep 10000 values then we can't remember variables to access them.

To resolve this problem we use arrays.

Arrays
=====

It refers to indexed collection of homogenous data elements.

Advantage of Arrays

1. we can represent multiple values by using single varaible, so that readability of the code will be improved.

eg::

```
 int arr[] =new int[1000];
```

we resolved the problem, but array is having limitation.

```
Student[] s=new Student[100];
 s[0] =new Student();
 s[1] =new Employee();//incompatible type: found Employee required:Student
```

===========================

To resolve this problem we can use

```
  Object[] obj =new Object[1000];
      obj[0]=new Student();
      obj[1]=new Employee();
```

Limitations of Arrays
=====================

1. Array is fixed in size, we can't increase or decrease the size of array.
2. To use the array compulsorily we should know the array size at the begining itself.
3. Array can hold only homogeneous datatype elements.
4. Array is not implemented using standard datastructure,so we don't have ready made methods to perform our task.
   eg: based on some condition, if we want to sort the student object in student[]
   direct methods are not available so it increases complexity of programmer.

To Overcome the limitations of Arrays we use "Collections".

Collections
==========

1. They are growable in nature(we can increase and decrease)
2. They can hold both hetereogenous and homogenous dataelements
3. Every collection class is implemented using some standard datastructure, so ready methods are available, as a programmer
   we need to implement rather we should just know how to call those methods.

Which one is prefered over Arrays and Collections?

Arrays is prefered, because performance is good.

Collections is not prefered because

1. List l=new ArrayList(); // default: 10 locations
   if 11th element has to added, then
   a. create a list with 11 locations
   b. copy all the elements from the previous collection
   c. copy the new reference into reference variable
   d. call garbage collector and clean the old memory.

Note:

=> To get something we need to compromise something, so if we use Collections performance is not upto the mark.

=> Array is language level concept(memory wise it is not good, perfomance is high)

=> Collection is API level(memory wise it is good,perfomance is low)

Difference b/w Arrays and Collection

```
=====================================
Arrays      =>  It is used only when Array size is fixed
Collection  =>  It is used only when size is not fixed(dynamic)

Arrays      =>  memory wise not recomended to use.
Collection  =>  memory wise  recomended to use.

Arrays      =>  Performance wise recomended to use.
Collection  =>  Perforamance wise it is not recomended to use.

Arrays      =>  It can hold only homogenous objects
Collection  =>  It can hold both heterogenous and homogenous Objects

Arrays      =>  We can hold both primitive values and Objects
                   eg: int[] arr=new int[5];
                   Integer[] arr=new Integer[5];
Collection  =>  It is capable of holding only objects not primitive types.

Arrays      =>  It is not implements using any standard datastructure, so no ready 
made methods
            for our requirement,it increases the complexity of programming.

Collection  =>  It is  implemented using  standard datastructure, so  ready made 
methods are
            available for our requirement,it is not complex.
```