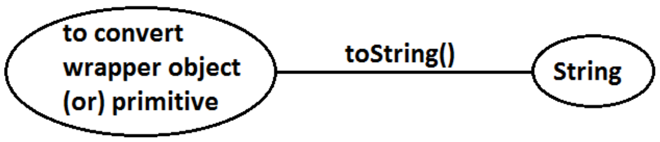
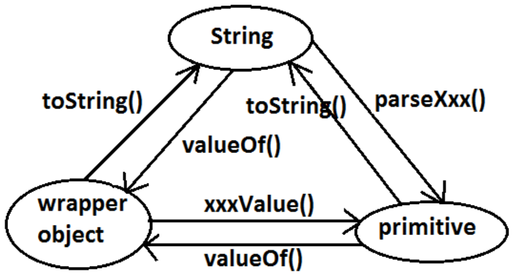
```
        System.out.println(s3);//14
        }
}
```

Diagram:



Dancing between String, wrapper object and primitive:

Diagram:



- String, StringBuffer, StringBuilder and all wrapper classes are final classes.
- The wrapper classes which are not child class of **Number** of Boolean and Character.
- The wrapper classes which are not direct class of Object of Byte, Short, Integer, Long, Float, Double.
- Sometimes we can consider **Void** is also as wrapper class.
- In addition to String all wrapper objects also immutable in java.

Autoboxing and Autounboxing

- Until 1.4 version we can't provide wrapper object in the place of primitive and primitive in the place of wrapper object all the required conversions should be performed explicitly by the programmer.

Example 1:

Program 1:

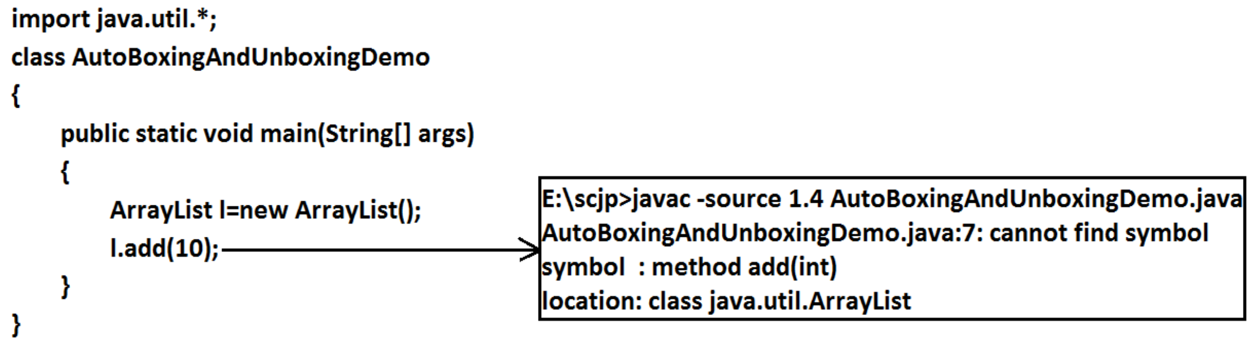
```
class AutoBoxingAndUnboxingDemo
{
    public static void main(String[] args)
    {
        Boolean b=new Boolean(true);
        if(b)
        {
            System.out.println("hello");
        }}}
```

1.4V

```
E:\scjp>javac -source 1.4 AutoBoxingAndUnboxingDemo.java
AutoBoxingAndUnboxingDemo.java:6: incompatible types
found   : java.lang.Boolean
required: boolean
```

Program 2:
```
class AutoBoxingAndUnboxingDemo
{
        public static void main(String[] args)
        {
                Boolean b=new Boolean(true);
                if(b)
                {
                        System.out.println("hello");
                }}}
```
Output:
```
Hello
```
Example 2:

Program 1:



```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
    public static void main(String[] args)
    {
        ArrayList l=new ArrayList();
        l.add(10);
    }
}
```
```
E:\scjp>javac -source 1.4 AutoBoxingAndUnboxingDemo.java
AutoBoxingAndUnboxingDemo.java:7: cannot find symbol
symbol  : method add(int)
location: class java.util.ArrayList
```
Program 2:
```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
        public static void main(String[] args)
        {
                ArrayList l=new ArrayList();
                Integer i=new Integer(10);
                l.add(i);
        }
}
```

- But from 1.5 version onwards we can provide primitive value in the place of wrapper and wrapper object in the place of primitive all required conversions will be performed automatically by compiler. These automatic conversions are called Autoboxing and Autounboxing.

Autoboxing: Automatic conversion of primitive to wrapper object by compiler is called Autoboxing.

Example:

Integer i=10; [compiler converts “int” to “Integer” automatically by Autoboxing]

- After compilation the above line will become.

Integer i=Integer.valueOf(10);

- That is internally Autoboxing concept is implemented by using valueOf() method.

Autounboxing: automatic conversion of wrapper object to primitive by compiler is called Autounboxing.

Example:

Integer i=new Integer(10);

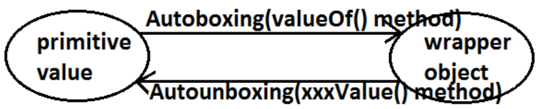
Int i=I; [compiler converts “Integer” to “int” automatically by Autounboxing]

- After compilation the above line will become.

Int i=I.intValue();

- That is Autounboxing concept is internally implemented by using xxxValue() method.

Diagram:



Example:

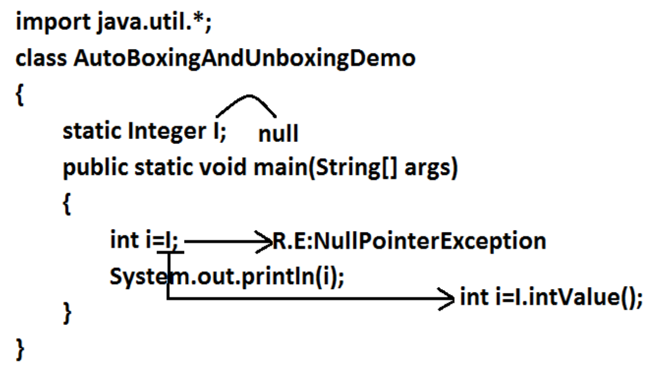
```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
      static Integer I=10;————① Autoboxing.
      public static void main(String[] args)
      {
            int i=I;————② Autounboxing
            methodOne(i);
                              ③ Autoboxing.
      }
      public static void methodOne(Integer I)
      {
            int k=I;————④ Autounboxing
            System.out.println(k);//10
      }
```

Note: From 1.5 version onwards we can use primitives and wrapper objects interchangly the required conversions will be performed automatically by compiler.

Example 1:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
        static Integer I=0;
        public static void main(String[] args)
        {
                int i=I;
                System.out.println(i);//0
        }
}
```

Example 2:

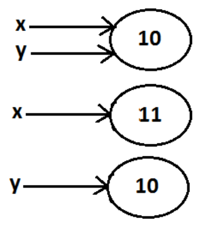

null
R.E:NullPointerException
int i=I.intValue();

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
        static Integer I;
        public static void main(String[] args)
        {
                int i=I;
                System.out.println(i);
        }
}
```

Example 3:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
        public static void main(String[] args)
        {
                Integer x=10;
                Integer y=x;
                ++x;
                System.out.println(x);//11
                System.out.println(y);//10
                System.out.println(x==y);//false
        }
}
```

Diagram:

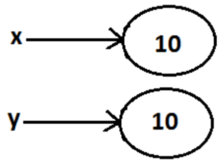


Note: All wrapper objects are immutable that is once we created a wrapper object we can't perform any changes in the existing object. If we are trying to perform any changes with those changes a new object will be created.

Example 4:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
	public static void main(String[] args)
	{
		Integer x=new Integer(10);
		Integer y=new Integer(10);
		System.out.println(x==y);//false
	}
}
```

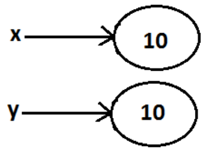
Diagram:



Example 5:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
	public static void main(String[] args)
	{
		Integer x=new Integer(10);
		Integer y=10;
		System.out.println(x==y);//false
	}
}
```

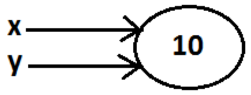
Diagram:



Example 6:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
        public static void main(String[] args)
        {
                Integer x=new Integer(10);
                Integer y=x;
                System.out.println(x==y);//true
        }
}
```

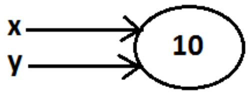
Diagram:



Example 7:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
        public static void main(String[] args)
        {
                Integer x=10;
                Integer y=10;
                System.out.println(x==y);//true
        }
}
```

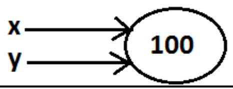
Diagram:



Example 8:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
        public static void main(String[] args)
        {
                Integer x=100;
                Integer y=100;
                System.out.println(x==y);//true
        }
}
```

Diagram:



Example 9:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
	public static void main(String[] args)
	{
		Integer x=1000;
		Integer y=1000;
		System.out.println(x==y);//false
	}
}
```

Diagram:

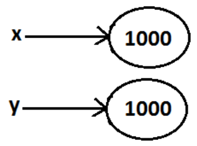
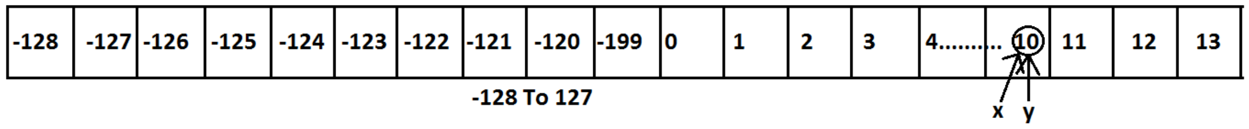


Diagram:



Conclusions:

- To implement the Autoboxing concept in every wrapper class a buffer of objects will be created at the time of class loading.
- By Autoboxing if an object is required to create 1st JVM will check whether that object is available in the buffer or not. If it is available then JVM will reuse that buffered object instead of creating new object. If the object is not available in the buffer then only a new object will be created. This approach improves performance and memory utilization.
- But this buffer concept is available only in the following cases.

Byte	Always
Short	-128 To 127
Integer	-128 To 127
Long	-128 To 127
Character	0 To 127
Boolean	Always

- In all the remaining cases compulsory a new object will be created.

Examples:

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

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

- Internally Autoboxing concept is implemented by using valueOf() method hence the above rule applicable even for valueOf() method also.

Examples:

```
① Integer x=new Integer(10);
  Integer y=new Integer(10);
  System.out.println(x==y);//false
```

```
② Integer x=10;
  Integer y=10;
  System.out.println(x==y);//true
```

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

```
④ Integer x=10;
  Integer y=Integer.valueOf(10);
  System.out.println(x==y);//true
```

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

Overloading with respect to widening, Autoboxing and var-arg methods:

Case 1: Widening vs Autoboxing.

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

Example:

```
import java.util.*;
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);
        }
}
```

<u>Output:</u>

```
Widening
```

- Widening dominates Autoboxing.

<u>Case 2:</u> Widening vs var-arg method.

<u>Example:</u>

```
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);
    }
}
```

<u>Output:</u>

```
Widening
```

- Widening dominates var-arg method.

<u>Case 3:</u> Autoboxing vs var-arg method.

<u>Example:</u>

```
import java.util.*;
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
```

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

    1) **Widening**
    2) **Autoboxing**
    3) **Var-arg method.**

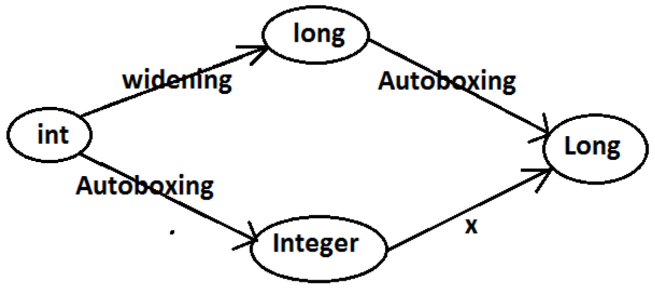
Case 4:

```
import java.util.*;
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 AutoBoxingAndUnb oxingDemo cannot be applied to (int)

Diagram:



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

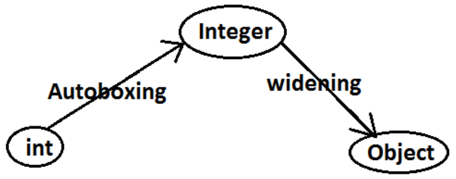
Case 5:

```
import java.util.*;
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
```

Diagram:



Which of the following declarations are valid?

1) Longl=10;(valid)
2) Long l=10;(invalid)(C.E)
3) Long l=10l;(Autoboxing)
4) Number n=10; (valid)
5) Object o=10.0; (valid)
6) int i=10l; (invalid)(C.E)