Java Lang Package

Topics

Module 14: Java Lang Package	
1.	Intro, Object class
2.	String class, Immutability concept
3.	StringBuffer class
4.	StringBuilder class
5.	Wrapper classes
6.	Autoboxing & AutoUnboxing

1. Introduction

- For writing any java program, whether it is simple or complex, the most commonly required classes & interfaces are grouped into a separate package which is nothing but java.lang package.
- We're not required to import java.lang package explicitly because all classes & interfaces present in lang package by default available to every java program.

Object class (java.lang.Object)

- The most commonly required methods for every java class (whether it is pre – defined class or customizer class) are defined in a separate class which is nothing but **Object** class.
- Every class in java is the child class of **Object** class either directly or indirectly so that **Object** class methods by default available to every java class. Hence **Object** class is considered as the root of all java classes.

- Object class defines the following 11 methods:

No.	Methods Name
1.	public String toString ()
2.	public native int hashCode ()
3.	public boolean equals (Object obj)
4.	protected native Object clone () throws CloneNotSupportedException
5.	protect void finalize () throws Throwable
6.	public final class getClass ()
7.	public final void wait () throws InterruptedException
8.	public final native void wait (long ms) throws InterruptedException
9.	public final void wait (long ms, int ns) throws InterruptedException
10.	public native final void notify ()
11.	public native final void notifyAll ()

Note: Strictly speaking, Object class contains 12 methods. The extra method is **registerNatives** () but this method is internally required for Object class & not available to the child classes.

1. toString () method

- We can use toString () method to get String representation of an Object.
- Whenever we're trying to print Object reference, internally toString () method will be called.

```
e.g.    Student s = new Student ();
        System.out.println(s);          →      System.out.println(s.toString());
```

- If our class does not contain toString () method then Object class toString () method will be executed which is implemented as follows

```
public String toString () {
    return getClass ().getName () + "@" + Integer.toHexString(hashCode());
}

i.e., classname@hashcode_in_hexadecimal_form      e.g. Student@1888759
```

- We can override toString () method to provide our own String representation.

```
E.g.    public String toString () {
               return name + "----" + rollno;
        }
```

- In all wrapper classes, in all Collection classes, String class, StringBuffer & StringBuilder classes, toString () method is overridden for meaningful string representation; hence it **is highly recommended to override toString () method in our class also**.

$$\begin{tikzcd}\boxed{\textbf{toString ()}} \arrow[d] \\ \boxed{\textbf{hashCode ()}}\end{tikzcd}$$

2. hashCode () method

- For every Object, a unique number is generated by JVM which is nothing but hashCode
- HashCode won't represent address of Object but hashCode will be generated based on the address of the Object.
- JVM will use HashCode while saving Objects into hashing related Data structures like Hashtable, HashMap, HashSet etc.
- The main advantage of saving Objects based on hashCode is search operation will become easy (The most powerful search algorithm up to today is Hashing (O (1))).
- Based on our requirement, we can override hashCode () method in our class to generate our own hashcode.
- Overriding hashCode () method is said to be proper if for every object a unique number as hashCode is generated.

- If we're giving the chance to Object class toString () method, it will internally call hashCode () method.
- If we're overriding toString () method then our toString () method may not call hashCode () method.

3. equals () method

- We can use equals () method to check equality of two objects**. E.g.,** obj1.equals(obj2);
- In String class, .equals() method is overridden for content comparison while In StringBuffer class, .equals() method is not overridden for content comparison.
- If our class does not contain equals () method then Object class equals () method will be executed which is meant for reference comparison (address comparison) i.e., if 2 references pointing to the same object then only .equals() method return true otherwise false.
- Based on our requirement, we can override equals () method for content comparison. While overriding equals () method for content comparison we have to take care about the following:
  a) What is the meaning of equality (i.e. whether we have to check only names or only rollno or both).
  b) If we're passing different type of Object, our equals () method should not rise **ClassCastException** i.e. we have to handle **ClassCastException** to return false.
  c) If we're passing null argument then our equals () method should not rise **NullPointerException** i.e. we have to handle **NullPointerException** to return false.

```
class Student {
   String name; int rollNo;
   Student (String name, int rollNo) {
      this.name = name; this.rollNo = rollNo;
   }
}
class DefaultEqualDemo {
   public static void main (String [] args) {
      Student s1 = new Student ("Sam", 11);
      Student s2 = new Student ("Will", 12);
      System.out.println(s1 == s2);               // false
      System.out.println(s1.equals(s2));       // false

      Student s3 = s1;
      System.out.println(s1 == s3);             // true
      System.out.println(s3.equals(s1));      // true

      Student s4 = new Student("Sam", 11);
      System.out.println(s4.equals(s1));          // false
      System.out.println(s4 == s1);                  // false
      System.out.println(s4.equals("Sam"));   // false
      System.out.println(s4.equals(null));      // false
   }
}
```

```
class Student {
   String name; int rollNo;
   Student (String name, int rollNo) {
      this.name = name; this.rollNo = rollNo;
   }
    public boolean equals (Object obj) {
      try {
         Student s = (Student)obj;
         if(name.equals(s.name) && rollNo == s.rollNo) return true;
         else return false;
      } catch (ClassCastException |   NullPointerException e) {
         return false;
      }
     }
}

class OveriddenEqualDemo {
   public static void main (String [] args) {
      Student s1 = new Student ("Sam", 11);
      Student s2 = new Student ("Will", 12);
      System.out.println(s1 == s2);               // false
      System.out.println(s1.equals(s2));       // false
      Student s3 = s1;
      System.out.println(s1 == s3);             // true
      System.out.println(s3.equals(s1));      // true
      Student s4 = new Student("Sam", 11);
      System.out.println(s4.equals(s1));          // true
      System.out.println(s4 == s1);                  // false
      System.out.println(s4.equals("Sam"));   // false
      System.out.println(s4.equals(null));      // false
   }
}
```

4. protected native Object clone () throws CloneNotSupportedException

- The process of creating exactly duplicate object is called Cloning.
- The main purpose of cloning is to maintain backup copy & to preserve the state of an object.
- We can perform cloning by using clone () method of Object class.

```
class Test implements Cloneable {
      int i = 10; int j = 20;
      public static void main (String [] args) throws CloneNotSupportedException {
         Test t1 = new Test ();
         Test t2 = (Test)t1.clone ();
         t2.i = 888; t2.j = 999;
         System.out.println(t1.i + " ----- " + t1.j);
         System.out.println(t2.i + " ----- " + t2.j);
      }
}
```

Output:

```
10 ----- 20
888 ----- 999
```

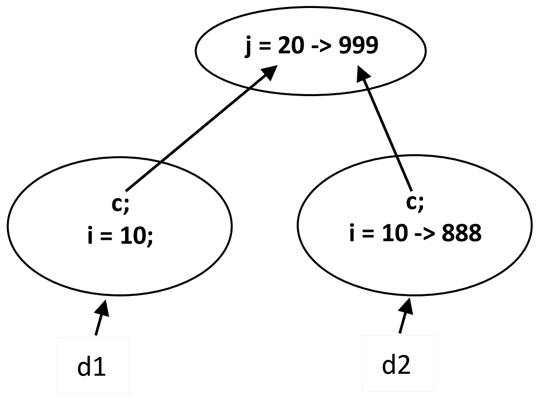
- We can perform cloning only for Cloneable objects. An object is said to be Cloneable iff the corresponding class implements Cloneable interface (market i/f)
- Cloneable i/f present in java.lang package & it doesn't contain any methods. It is a marker i/f.
- If we're trying to perform cloning for non – cloneable objects then we will get Runtime exception saying CloneNotSupportedException.
- 2 types of Cloning
  a. Shallow Cloning
  b. Deep Cloning

a) Shallow Cloning

- The process of create bitwise copy of an object is called Shallow cloning.
- If the main object contains primitive variables, then exactly duplicate copies will be created in the cloned object.
- **If the main object contains any reference variable, then corresponding object won't be created.**
- Object class clone () method meant for shallow cloning.

```
class Cat {
   int j;
   Cat (int j) {
      this.j = j;
   }
}

class Dog implements Cloneable {
   Cat c;
   int i;
   Dog (Cat c, int i) {
      this.c = c; this.i = i;
   }
   public Object clone () throws CloneNotSupportedException {
      return super.clone();
   }
}
```



```
class ShallowCloningDemo {
   public static void main (String [] args) throws CloneNotSupportedException {
      Cat c = new Cat (20);
      Dog d1 = new Dog (c, 10);
      System.out.println("Before Cloning; d1.i = " + d1.i + "; d1.c.j = " + d1.c.j);
      Dog d2 = (Dog)d1.clone();
      d2.i = 888; d2.c.j = 999;
      System.out.println("After Cloning; d1.i = " + d1.i + "; d1.c.j = " + d1.c.j);
      System.out.println("After Cloning; d2.i = " + d2.i + "; d2.c.j = " + d2.c.j);
      System.out.println("d1 = " + d1 + "; d2 = " + d2);
   }
}
```

Output:

```
Before Cloning; d1.i = 10; d1.c.j = 20
After Cloning; d1.i = 10; d1.c.j = 999
After Cloning; d2.i = 888; d2.c.j = 999
d1 = Dog@61baa894; d2 = Dog@b065c63
```

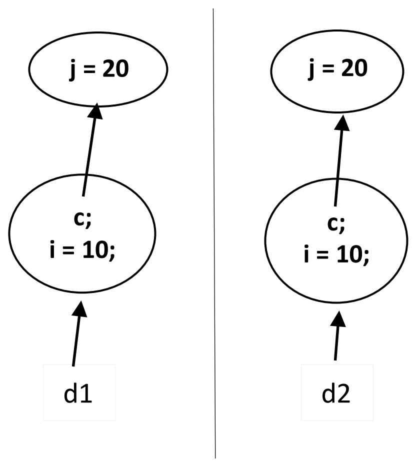
- In shallow cloning, by using cloned object reference if we perform any change to the contained object then those changes will be reflected to the main object.
- To overcome this problem, we should go for Deep cloning.

b) Deep Cloning

- The process of creating exactly duplicate independent copy including contained object is called Deep cloning.
- In deep cloning, if the main object contains any primitive variables, then duplicate copies of those variables will be created in the cloned object.
- If the main object contains any reference variables, then the corresponding contained object will also be created in the cloned copy.
- By default, Object class clone () method is meant for Shallow cloning. We can implement deep cloning explicitly by overriding clone () method in our class.

```
class Cat {
   int j;
   Cat (int j) {
      this.j = j;
   }
}

class Dog implements Cloneable {
   Cat c; int i;
   Dog(Cat c, int i) {
      this.c = c;
      this.i = i;
   }
   public Object clone () throws CloneNotSupportedException {
      Cat c1 = new Cat (c.j);
      Dog d = new Dog (c1, i);
      return d;
   }
}
```



```
class DeepCloningDemo {
      public static void main (String [] args) throws CloneNotSupportedException {
         Cat c = new Cat (20);
         Dog d1 = new Dog (c, 10);
         System.out.println("Before cloning: [d1.i = " + d1.i + "; d1.c.j = " + d1.c.j + "]");
         Dog d2 = (Dog)d1.clone();
         d2.i = 888; d2.c.j = 999;
         System.out.println("After cloning: [d1.i = " + d1.i + "; d1.c.j = " + d1.c.j + "]");
         System.out.println("After cloning: [d2.i = " + d2.i + "; d2.c.j = " + d2.c.j + "]");
      }
}

Output:
Before cloning: [d1.i = 10; d1.c.j = 20]
After cloning: [d1.i = 10; d1.c.j = 20]
After cloning: [d2.i = 888; d2.c.j = 999]
```

- By using cloned object reference, if we perform any change to the contained object, then those changes won't be reflected to main object.

Shallow Cloning Vs Deep Cloning

- If object contains only Primitive variables, then **Shallow Cloning** is the best choice while If object contains reference variables, then **Deep Cloning** is the best choice.

5. finalize () method

- Just before destroying an Object, Garbage Collector calls finalize () method to perform cleanup activities.
- Once finalize () method completes, automatically Garbage collector destroys that object.

6. getClass () method

- We can use getClass () method to get runtime class definition of an object.

```
public final Class getClass ()
```

- By using this class, Class object we can access class level properties like Fully Qualified Name of the class, Methods Information, Constructors info etc.

```
import java.lang.reflect.*;

class GetClassDemo {
    public static void main (String [] args) {
       int count = 0;
       Object o = new Object ();
       Class c = o.getClass();
       System.out.println("Fully Qualified name of the class: " + c.getName());
       Method [] m = c.getDeclaredMethods();
       System.out.println("Methods info: ");
       for (Method m1 : m) {
          count++;
          System.out.println(m1.getName());
       }
       System.out.println("The no. of methods in String class: " + count);          // 12
    }
}
```

Note:

1. After loading every .class file, JVM will create an object of the type java.lang.Class class.
2. Programmer can use this Class object to get class level information.
3. We can use getClass() method very frequently in reflection.

Refer Multithread notes for these methods.	
7.	public final void wait () throws InterruptedException
8.	public final native void wait (long ms) throws InterruptedException
9.	public final void wait (long ms, int ns) throws InterruptedException
10.	public native final void notify ()
11.	public native final void notifyAll ()

2. String

String

1. Once we create a String object, we can't perform any changes in the existing object. If we're trying to perform any change then with those changes, a new object will be created. This non – changeable behavior is nothing but Immutability of String.

No.	String	StringBuffer

1.	Once we create a String object, we can't perform any changes in the existing object. If we're trying to perform any change then with those changes, a new object will be created. This non – changeable behavior is nothing but Immutability of String.
	E.g., String s = new String ("Sam");
	s.concat("Will");
	SoPln(s); // Sam
	s = s.concat(" Smith");
	SoPln(s); // Sam Smith
		Once we create a StringBuffer object, we can perform any change in the existing object. This changeable behavior is nothing but Mutability of StringBuffer object.
		
		E.g., StringBuffer sb = new StringBuffer("Sam");
		sb.append(" will");
		SoPln(sb); // Sam Will

2.	In String class, .equals() method is overridden for content comparison hence even though objects are different, if content is same .equals() method returns true.
	
	e.g., String s1 = new String("Sam");
	String s2 = new String("Sam");
	SoPln (s1 == s2); // **false**
	SoPln (s1.equals(s2)); // **true**
		In StringBuffer class, .equals() method is not overridden for content comparison hence Object class .equals() method gets executed which is meant for reference comparison.
		e.g., StringBuffer sb1 = new StringBuffer("Sam");
		StringBuffer sb2 = new StringBuffer("Sam");
		SoPln (sb1 == sb2); // **false**
		SoPln (sb1.equals(sb2)); // **true**

Notes:

1. Object creation in String constant pool (SCP) is optional. First JVM will check if any object is already present in SCP with required content, if object already present then existing object will be reused. If not present, then a new object will be created. (This rule is applicable only for SCP not for Heap)

String s = new String ("Sam")	**String s = "Sam";**
In this case, **2 objects will be created**. One in the heap area & other in SCP. Reference variable s will always point to heap object.	In this case, only 1 object will be created in SCP & s will point to that object in SCP.

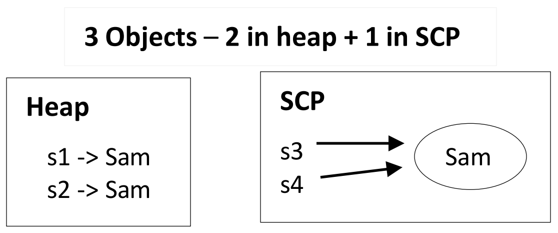
Heap	**SCP**
s -> Sam	Sam

Heap	**SCP**
	Sam

2. Garbage collector is not allowed to access SCP area, hence even though object doesn't contain reference variable, it is not eligible for Garbage collection. Garbage collection only applicable for Heap.
3. All SCP objects will be destroyed automatically at the time of JVM shutdown.
4. Whenever we're using new operator, compulsory a new Object will be created in the Heap area, hence there may be a chance of existing 2 objects with same content in the heap area but not in SCP i.e., duplicate objects are possible in the heap area but not in SCP area.
5. For every String constant, one object will be placed in SCP area.
6. Because of some runtime operation, if an object is required to create, that object will be placed only in the heap area but not in SCP area.

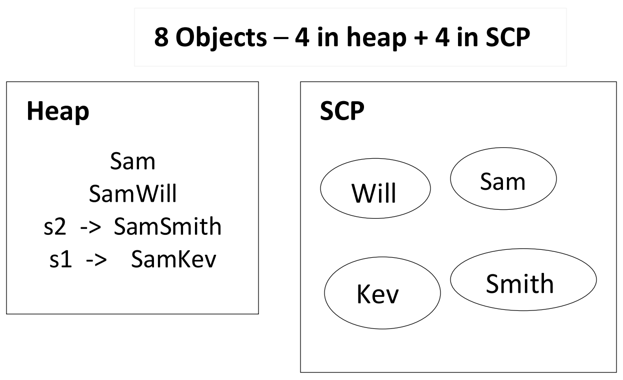
E.g.,1

```
String s1 = new String("Sam");
String s2 = new String("Sam");
String s3 = "Sam";
String s4 = "Sam";
System.out.println(s1 == s2);    // false
System.out.println(s1 == s3);    // false
System.out.println(s3 == s4);    // true
```



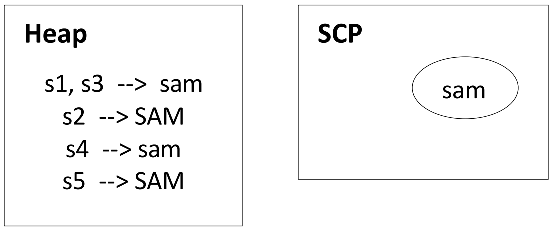
E.g.,2

```
String s1 = new String ("Sam");
s1.concat ("Will");
SoPln ("s1: " + s1);         // Sam
String s2 = s1.concat ("Smith");
SoPln ("s2: " + s2);         // SamSmith
s1 = s1.concat ("Kev");
SoPln ("s1: " + s1);        //  SamKev
```



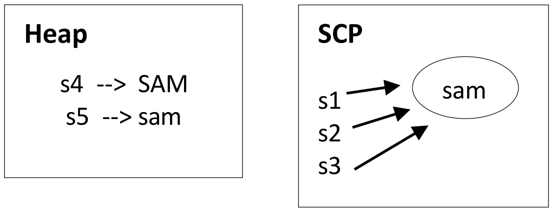
E.g.,3

```
String s1 = new String ("sam");
String s2 = s1.toUpperCase();
String s3 = s1.toLowerCase();
SoPln (s1 == s2);    // false
SoPln (s2 == s3);   //  true
String s4 = s2.toLowerCase();
String s5 = s4.toUpperCase();
SoPln (s3 == s4);    // false
SoPln (s2 == s5);    // false
```



E.g.,4

```
String s1 = "sam";
String s2 = s1.toString();
String s3 = s1.toLowerCase();
String s4 = s1.toUpperCase();
String s5 = s4.toLowerCase();
```



Constructors of String class

1. String s = new String ();
2. String s = new String (String literal);
3. String s = new String (StringBuffer sb);
4. String s = new String (char [] ch);
5. String s = new String (byte [] b);

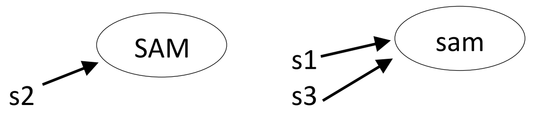
Imp. Methods of String class

No.	Methods	Description
1.	char charAt (int index)	Returns the character locating at specified index.
2.	String concat (String s)	The overloaded + & += operators also meant for concatenation purpose
3.	boolean equals (Object o)	To perform content comparison where case is important
4.	boolean equalsIgnoreCase (String s)	To perform content comparison when case is not important
5.	String substring (int begin)	Returns substring from begin index to end of the String
6.	String substring (int begin, int end)	Returns substring from begin index to end – 1 index (end exclusive)
7.	int length ()	Returns no. of characters present in the string
8.	String replace (char oldCh, char newCh)	
9.	String toLowerCase ()	
10.	String toUpperCase ()	
11.	String trim ()	To remove blankspaces present at beginning & end of the string but not middle blankspaces
12.	int indexOf (char ch)	Returns the index of 1st occurrence of specified character
13.	int lastOf (char ch)	

How to create our Own Immutable class

- Once we create an object we can't perform any changes in that object. If we're trying to perform any change & if there is a change in the content then with those changes a new object will be created.
- If there is no change in the content then existing object will be reused. This behavior is nothing but Immutability.

```
String s1 = new String("sam");
String s2 = s1.toUpperCase();
String s3 = s1.toLowerCase();
```



```
class ImmutabilityDemo {
   public static void main(String[] args) {
      Test t1 = new Test(10);
      Test t2 = t1.modify(100);
      Test t3 = t1.modify(10);
      System.out.println(t1 == t2);     // false
      System.out.println(t1 == t3);     // true

   }
}

final class Test {
   private int i;
   Test (int i) {
      this.i = i;
   }
   public Test modify (int i) {
      if(this.i == i)
         return this;
      return new Test(i);
   }
}
```

final Vs Immutability

No.	final modifier (final variable)	Immutability concept (Immutable object)
1.	final modifier is applicable for variables but not for objects.	Immutability concept is applicable for objects but not for variables.
2.	By declaring a variable final, we can't perform any change in the variable.	By declaring a reference variable as final, we won't get any immutability nature even though we can perform any type of change in the corresponding object, but we can't perform reassignment for that variable.

3.		e.g., final StringBuffer sb = new StringBuffer("sam");
		sb.append ("smith");
		SoPln (sb); // samsmith
		sb = new StringBuffer("kevin"); // Compiler error

4. StringBuffer class

- If the content is fixed & won't change frequently then it is recommended to go for String.
- If the content is not fixed & keep on changing then it's not recommended to use String because for every change, a new object will be created which affects performance of the system.
- To handle this requirement, we should go for StringBuffer.
- The main adv. of StringBuffer over String is all required changes will be performed in the existing object only.

Constructors

1. StringBuffer sb = new StringBuffer ();
   - Creates an empty StringBuffer object with default initial capacity as 16.
   - Once StringBuffer reaches its max capacity, a new StringBuffer object will be created with

     $\boxed{\text{New\_capacity = (Current\_capacity + 1) * 2}}$

     e.g., 16 → 34 → 70 etc.

2. StringBuffer sb = new StringBuffer (int initial_capacity);
3. StringBuffer sb = new StringBuffer (String s);
   - Creates an equivalent StringBuffer for the given String with capacity $\boxed{\text{capacity = (str.length() + 16)}}$

Imp. Methods of StringBuffer class

No.	Methods	Description
1.	int length ()	No. of characters already present in StringBuffer
2.	int capacity ()	Total how many characters a StringBuffer can accommodate. Default initial capacity is 16
3.	char charAt (int index)	
4.	void setCharAt (int index, char ch)	
5.	StringBuffer append(String s) StringBuffer append(int i) StringBuffer append(long l) StringBuffer append(char ch) StringBuffer append(boolean b) ………. overloaded methods	
6.	StringBuffer insert (int index, String s) StringBuffer insert (int index, int i) StringBuffer insert (int index, long l) StringBuffer insert (int index, char ch) StringBuffer insert (int index, boolean b) ……….overloaded methods	
7.	StringBuffer delete (int begin, int end)	Delete characters located from begin to end – 1 index
8.	StringBuffer deleteCharAt (int index)	
9.	StringBuffer reverse ()	
10.	void setLength (int length)	Cut down length of StringBuffer to specified length.
11.	void ensureCapacity (int capacity)	To increase capacity on fly based on our requirement
12.	void trimToSize()	To deallocate extra allocated free memory

5. StringBuilder class

- Every method present in StringBuffer class is synchronized & hence only one thread is allowed to operate on StringBuffer object at a time which may create performance problems.
- To handle this requirement, SUN people introduced StringBuilder concept in 1.5 version
- StringBuilder is exactly same as StringBuffer except the following differences:

No.	StringBuffer	StringBuilder
1.	Every method present in StringBuffer is synchronized	Every method present in StringBuilder is non – synchronized.
2.	At a time, only one thread is allowed to operate on StringBuffer object & hence, StringBuffer object is thread – safe.	At a time, multiple threads are allowed to operate on StringBuffer object & hence StringBuffer is not thread safe.
3.	Threads are required to wait to operate on StringBuffer object & hence relatively performance is low.	Threads are not required to wait to operate on StringBuilder object & hence relatively performance is high.
4.	Introduced in 1.0 v	Introduced in 1.5 v

- Except the above differences, everything is same in StringBuffer & StringBuilder (including Methods & constructors)
- String Vs StringBuffer Vs StringBuilder
  - If the content is fixed & won't change frequently then we should go for **String**.
  - If the content is not fixed & keep on changing but thread – safety is required, then we should go for **StringBuffer**.
  - If the content is not fixed & keep on changing but thread – safety is not required then we should go for **StringBuilder**.

Method Chaining

- For most of the methods in String, StringBuffer & StringBuilder, return types are same type, hence after applying a method on the result we can call another method which forms method chaining.
- In method chaining, method calls will be executed from left to right.

```
sb.m1 ().m2 ().m3 ()…….
```

```
StringBuffer sb = new StringBuffer ();
sb.append("abc").append("def").insert(2, "xyz").delete(2,4).reverse();
System.out.println(sb);      // abcdef -> abxyzcdef -> abzcdef -> fedczba
```

6. Wrapper Classes

- The main objectives of Wrapper classes are:
  a. To wrap primitive into object form so that we can handle primitives also just like Objects.
  b. To define several utility methods which are required for the primitives.

Constructors

- Almost all wrapper classes contain 2 constructors:
  a. One can take corresponding primitive as argument.
  b. Other can take String as argument.

```
Integer I = new Integer (10);
Integer I = new Integer ("10");

If the String argument is not representing a number, then we will get RE: NumberFormatException
Integer I = new Integer ("ten");      // RE: NumberFormatException

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

- Float class contains 3 constructors with float, double & String arguments.
- Character class contains only one constructor which can take char argument.
- Boolean class contains 2 constructors.
  - If we pass boolean primitive as argument, the only allowed values are true & false where case & content is important.
  - If we are passing String type as argument then case & content both are not important. If the content is case insensitive String of "true" then it's treated as true otherwise it's treated as false.

```
Boolean b = new Boolean (true);   // true
Boolean b = new Boolean (false);   // false
Boolean b = new Boolean (True);   // Wrong
Boolean b = new Boolean (False);  // Wrong
```

```
Boolean b = new Boolean ("true");          // true
Boolean b = new Boolean ("True");          // true
Boolean b = new Boolean ("TRUE");          // true
Boolean b = new Boolean ("malika");        //  false
Boolean b = new Boolean ("jareena");       //  false
```

- In all wrapper classes, toString () method is overridden to return content directly.
- In all wrapper classes, .equals () method is overridden for content comparison.

No.	Wrapper classes	Corresponding Constructor Arguments
1.	Byte	byte or String
2.	Short	short or String
3.	Integer	int or String
4.	Long	long or String
5.	Float	float or String or double
6.	Double	double or String
7.	Character	Char
8.	Boolean	boolean or String

Various utility methods present in Wrapper classes

No.	**Utility Methods (Pattern)**	**Utility Methods**

1.	**valueOf ()**	We can use valueOf () methods to create wrapper object for the given Primitive or String.
		
		Form 1: Every wrapper class except Character class contains static valueOf () method to create wrapper object for the given String.
		
		public static Wrapper **valueOf (**String s**)**
		E.g., Integer I = Integer.valueOf (“10”);
		Double d = Double.valueOf (“10.5”);
		Boolean b = Boolean.valueOf (“sam”);
		
		Form 2: Every integral wrapper class (Byte, Short, Integer, Long) contains the following value of method to create wrapper object for the given specified radix string. The allowed range of radix is 2 to 36.
		
		public static Wrapper **valueOf (**String s, int radix**)**
		E.g., Integer I = Integer.valueOf (“100”, 2); // 4
		Integer I = Integer.valueOf (“101”, 4); // 17
		
		Form 3: Every wrapper class including Character class contains a static valueOf () method to create wrapper object for the given primitive.
		
		public static Wrapper **valueOf (**primitive p**)**
		E.g., Integer I = Integer.valueOf (10);
		Character c = Character.valueOf (‘a’);
		Boolean b = Boolean.valueOf (true);

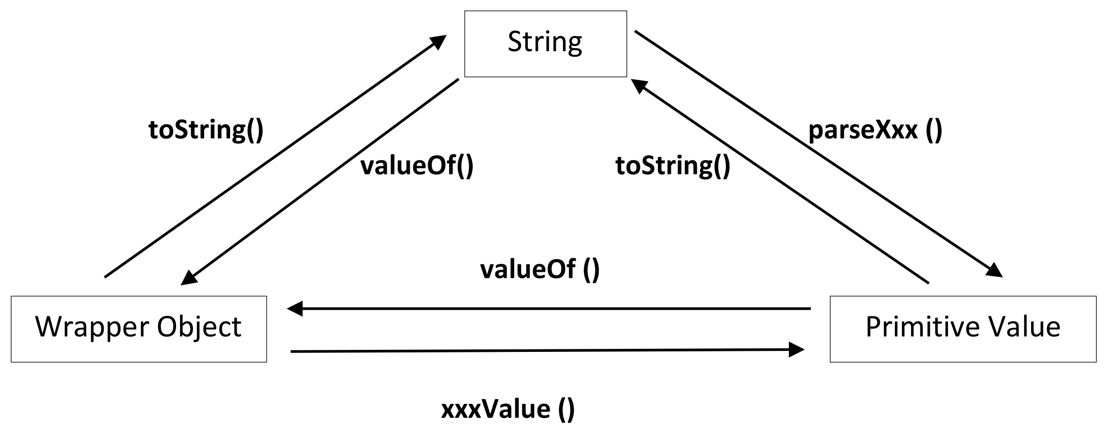
2.	**xxxValue ()**	We can use xxxValue () methods to get primitive for the given wrapper object.
		
		Every number type wrapper class (Byte, Short, Integer, Long, Float, Double) contains the following 6 methods to get Primitive for the given wrapper object.
		I) byte byteValue ();
		II) short shortValue ();
		III) int intValue ();
		IV) long longValue ();
		V) float floatValue ();
		VI) double doubleValue ();
		VII) char charValue ();
		VIII) boolean booleanValue ();

		Integer I = new Integer (130);
		SoPln (I.byteValue()); **// -126**
		SoPln (I.shortValue()); **// 130**
		SoPln (I.intValue()); **// 130**
		SoPln (I.longValue()); **// 130**
		SoPln (I.floatValue()); **// 130.0**
		SoPln (I.doubleValue()); **// 130.0**
		
		Character ch = new Character (‘b’);
		char c = ch.charValue ();
		SoPln (c); **// a**
		
		Boolean B = new Boolean (“sam”);
		boolean b = B.booleanValue ();
		SoPln (b); **// false**

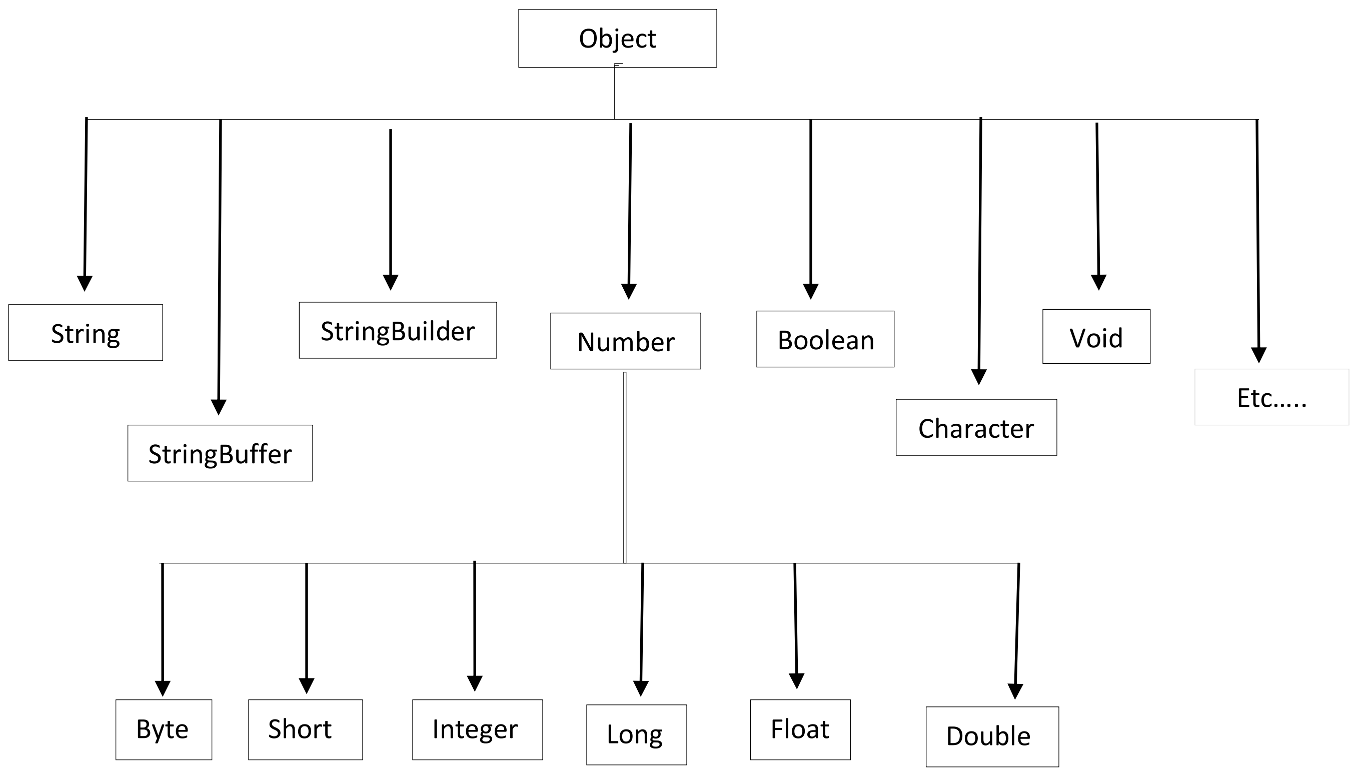
3.	**parseXxx ()**	We can use parseXxx () methods to convert string to Primitive. **Form 1:** Every wrapper class except Character class contains the following **parseXxx ()** method to find primitive for the given String object public static primitive **parseXxx** (String s) **E.g.,** int i = Integer.parseInt (“10”); double d = Double.parseDouble (“10.5”); boolean b = Boolean.parseBoolean (“true”); **Form 2:** Every integral type wrapper class (Byte, Short, Integer, Long) contains the following **parseXxx ()** method to convert specified radix string to Primitive (Allowed range of radix is 2 to 36) public static primitive **parseXxx** (String s, int radix) **E.g.,** int i = Integer.parseInt (“111”, 2); SoPln (i); // 15

4.	**toString ()**	We can use toString () method to convert wrapper object or Primitive to String.
		
		Form 1: Every wrapper class contains the following toString () method to convert wrapper object to String type.
		• It is the overriding version of Object class toString () method.
		• Whenever we're trying to print Wrapper object reference, internally toString () method will be called.
		
		public String **toString** ()
		E.g., Integer I = new Integer (10);
		String s = I.**toString**();
		SoPln (s); // 10
		
		Form 2: Every wrapper class including Character class, contains the following static toString () method to convert primitive to String.
		
		public static String **toString** ()
		E.g., String s = Integer.**toString** (10);
		String s = Boolean.**toString** (true);
		String s = Character.**toString** ('a');
		
		Form 3: Integer & Long classes contain the following toString () method to convert Primitive to specified radix string.
		
		public static String **toString** (primitive p, int radix)
		E.g., String s = Integer.**toString** (15, 2);
		SoPln (s); // 1111
		
		Form 4: Integer & Long classes contain the following toXxxString () methods
		
		public static String **toBinaryString** (primitive p)
		public static String **toOctalString** (primitive p)
		public static String **toHexString** (primitive p)
		
		E.g., String s = Integer.toBinaryString (10);
		SoPln (s); **// 1010**
		String s = Integer.toOctalString (10);
		SoPln (s); **// 12**
		String s = Integer.toHexString (10);
		SoPln (s); **// a**

Dancing b/w String, Wrapper Object, Primitive



Partial Hierarchy of java.lang package



- In addition to String objects, all wrapper class objects are also Immutable.
- Sometimes Void class is also considered as Wrapper class.

Void class

- It is a final class & it is the direct child class of Object.
- It doesn't contain any methods & it contains only one variable **Void.TYPE**
- In General, we can use Void class in reflections to check whether the method return type is void or not.
- Void is a class representation of void keyword in Java

```
if (getMethod ("m1") . getReturnType () == Void.TYPE) { …. }
```

7. AutoBoxing & AutoUnboxing

AutoBoxing

- Automatic conversion of Primitive to wrapper object by compiler is called Autoboxing.
- Autoboxing concept is implemented by using valueOf () methods.

`Integer I = 10;` $\longrightarrow$ `Integer I = Integer.valueOf(10)`

AutoUnboxing

- Automatic conversion of wrapper object to Primitive by compiler is called AutoUnboxing.
- Internally, AutoUnboxing concept is implemented by using xxxValue () methods.

`Integer I = new Integer (10);` $\longrightarrow$ `int i = I.intValue(10)`
`int i = I`

$$\begin{tikzcd}\boxed{\text{Primitive Value}} \arrow[r, shift left, "\textbf{valueOf ()}\ \text{[Autoboxing]}"] & \boxed{\text{Wrapper Object}} \arrow[l, shift left, "\textbf{xxxValue ()}\ \text{[AutoUnboxing]}"]\end{tikzcd}$$

```
class AutoBoxingUnBoxingDemo {
  static Integer I = 10;
  public static void main (String [] args) {
    int i = I;                                    // AutoUnboxing
    m1(i);                                        // AutoBoxing
  }

  public static void m1 (Integer K) {             // AutoBoxing
    int m = K;                                    // AutoUnboxing
    System.out.println(m);                        // 10
  }
}
```

Notes:

- Internally, to provide support for Autoboxing, a Buffer of wrapper objects will be created at the time of Wrapper class loading.
- By Autoboxing, if an object is required to create first JVM will check whether this object already present in the buffer or not.
- If it's already present in the buffer then existing buffer object will be used. If not already present in the buffer, then JVM will create a new object.
- Buffer concept is available only in the below ranges. Except this range, in all remaining cases a new object will be created.
- Internally, Autoboxing concept is implemented by using **valueOf ()** methods, hence buffering concept is applicable for valueOf () methods also.

No.	Wrapper classes	Range
1.	Byte	Always
2.	Short	-128 to 127
3.	Integer	-128 to 127
4.	Long	-128 to 127
5.	Character	0 to 127
6.	Boolean	Always
Float & Double not applicable		

Integer x = 127;
Integer y = 127;
SoPln (x == y);   // true
	Integer x = 128;
	Integer y = 128;
	SoPln (x == y); // false
		Double x = 10.0;
		Double y = 10.0;
		SoPln (x == y); // false
			Boolean x = false;
			Boolean y = false;
			SoPln (x == y); // true

Overloading wrt Autoboxing, widening & var – arg methods

Case 1: Widening dominates Autoboxing

```
class Test {
   public static void m1(long l) {
      System.out.println("Widening");
   }
   public static void m1(Integer i) {
      System.out.println("Autoboxing");
   }
   public static void main (String [] args) {
      int x = 10;
      m1 (x);
   }
}
```

Output:

```
Widening
```

Case 2: Widening dominates var – arg methods

```
class Test {
   public static void m1(long l) {
      System.out.println("Widening");
   }
   public static void m1(int... x) {
      System.out.println("var-arg method");
   }
   public static void main (String [] args) {
      int x = 10;
      m1 (x);
   }
}
```
Output:
```
Widening
```

Case 3: Autoboxing dominates var – arg method

```
class Test {
   public static void m1(int... x) {
      System.out.println("var-arg method");
   }
   public static void m1(Integer i) {
      System.out.println("Autoboxing");
   }
   public static void main (String [] args) {
      int x = 10;
      m1 (x);
   }
}
```
Output:
```
Autoboxing
```

- While resolving overloaded methods, compiler will always give the precedence in the following order
  Widening >>> **Autoboxing** >>> **var – arg methods**

Case 4: Widening followed by Autoboxing is not allowed in Java whereas Autoboxing followed by widening is allowed.

```
class Test {
   public static void m1(Long l) {
      System.out.println("Widening");
   }
   public static void main (String [] args) {
      int x = 10;
      m1 (x);
   }
}
```
Output:

```
ERROR – incompatible types: int cannot be converted to Long
```

Case 5: Autoboxing followed by widening is allowed.

```
class Test {
   public static void m1(Object o) {
      System.out.println("Object version");
   }
   public static void main (String [] args) {
      int x = 10;
      m1 (x);
   }
}
```
Output:

```
Object version
```

Q. Which of the following assignments are legal

int i = 10;	// primitive
Integer I = 10;	// Autoboxing
int i = 10L;	// Wrong
Long l = 10L;	// Autoboxing
Long l = 10;	// CE: incompatible types
long l = 10;	// Widening
Object obj = 10;	// Autoboxing -> Widening
double d = 10;	// Widening
Double d = 10;	// CE: incompatible types
Number n = 10;	// Autoboxing -> Widening