| **No.** | **Methods** | **Description** |
| --- | --- | --- |
| 1 | char s1.charAt(int index) | returns char value for the particular index |
| 2 | int s1.length() | returns string length |
| 3 | String s1.substring(int beginIndex) | returns substring for given begin index. |
|  | String s1.substring(int beginIndex, int endIndex) | returns substring for given begin index(inclusive) and end index(exclusive). |
| 4 | boolean s1.contains(CharSequence s) | returns true or false after matching the sequence of char value. |
| 5 | boolean s1.equals(Object another) | checks the equality of string with the given object. |
| 6 | boolean s1.isEmpty() | checks if string is empty. |
| 7 | String s1.concat(String str) | concatenates the specified string. |
| 8 | String s1.replace(char old, char new) | replaces all occurrences of the specified char value. |
|  | String s1.replace(CharSequence old, CharSequence new) | replaces all occurrences of the specified CharSequence. |
| 9 | String s1.equalsIgnoreCase(String another) | compares another string. It doesn't check case. |
| 10 | String[] s1.split(String regex) | returns a split string matching regex. |
|  | String[] s1.split(String regex, int limit) | returns a split string matching regex and limit. |
| 11 | int s1.indexOf(int ch) | returns the specified char value index. |
|  | int s1.indexOf(int ch, int fromIndex) | returns the specified char value index starting with given index. |
|  | int s1.indexOf(String substring) | returns the specified substring index. |
|  | int s1.indexOf(String substring, int fromIndex) | returns the specified substring index starting with given index. |
| 12 | String s1.toLowerCase() | returns a string in lowercase. |
| 13 | String s1.toUpperCase() | returns a string in uppercase. |
| 14 | String s1.trim() | removes the beginning and ending spaces of this string. |
| 15 | String String.valueOf(int value) | converts the given type into string. It is an overloaded method. |

**Math Class**

If the return type is of the type same as an argument then we are using *var*

| **No.** | **Methods** | **Description** |
| --- | --- | --- |
| 1 | var Math.abs(a) | return the Absolute value of the given value. |
| 2 | var Math.max(a,b) | returns the Largest of two values. |
| 3 | long Math.round(float/double x) | round off the decimal numbers to the nearest value. |
| 4 | double Math.sqrt(double x) | return the square root of a number. |
| 5 | double Math.cbrt(double x) | return the cube root of a number. |
| 6 | double Math.pow(double a, double b) | returns the value of first argument raised to the power to second argument. |
| 7 | double Math.signum(a) => (a>0): 1.0 / (a<0): -1.0 / (a==0): 0 | used to find the sign of a given value. |
| 8 | double Math.ceil(double x) | used to find the smallest integer value that is greater than or equal to the argument or mathematical integer. |
| 9 | double Math.floor(double a) | used to find the largest integer value which is less than or equal to the argument |
| 10 | double Math.random() | returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0. |
| 11 | double Math.log(double x) | returns the natural logarithm. |
| 12 | double Math.log10(double x) | return the base 10 logarithm of a double value. |
| 13 | double Math.exp(double x) | returns E raised to the power of a double value |
| 14 | double Math.sin/cos/tan/asin/acos/atan(double a) | return the trigonometric value. |
| 15 | double sinh/tanh/cosh(double x) | return the trigonometric Hyperbolic value. |

**Integer Class**

| **No.** | **Methods** | **Description** |
| --- | --- | --- |
| 1 | int Integer.compare(int x,int y) | compare two int x>y => 1 \ x=y => 0 \ x<y => -1 |
| 2 | int x.compareTo(int x) | compare two int x>y => 1 \ x=y => 0 \ x<y => -1 |
| 3 | boolean Integer.equals(Object obj) | compares the value of the parameter to the value of the current Integer object and returns boolean ( True or False ). |
| 4 | int object.intValue() | returns the value of the specified number as an int. |
| 5 | long object.longValue() | returns the value of the specified number as an long. |
| 6 | int Integer.parseInt (String s) | parses the String argument as a signed decimal integer object |
|  | int Integer.parseInt (String s, int radix) | parses the String argument as a signed decimal integer object in the specified radix by the second argument |
|  | int Integer.parseInt (CharSequence s,int beginIndex,int endIndex,int radix) | parses the CharSequence argument as a signed integer in the specified radix argument, beginning at the specified beginIndex and extending to endIndex - 1. |
| 7 | int Integer.reverse(int i) | method returns the numeric value obtained by reversing order of the bits in the specified int value. |
| 8 | int Integer.rotateLeft(int i, int distance) | returns the value obtained by rotating the two's complement binary representation of the specified int value left by the specified number of bits. |
|  | int Integer.rotateRight(int i, int distance) | returns the value obtained by rotating the two's complement binary representation of the specified int value right by the specified number of bits. |
| 9 | int Integer.signum(int i) | (a>0): 1.0 / (a<0): -1.0 / (a==0): 0 |
| 10 | String Integer.toBinaryString (int i) | returns a string representation of the integer argument as an unsigned integer in binary base 2. |
|  | String Integer.toHexString (int i) | returns a string representation of the integer argument as an unsigned integer in binary base 16. |
|  | String Integer.toOctalString (int i) | returns a string representation of the integer argument as an unsigned integer in binary base 8. |
| 11 | String a.toString() | returns a String object representing the value of the Number Object.(int/float/double/boolean etc) |
|  | String Integer.toString(int i) | returns a string representation of the int type argument in base 10. |
|  | String Integer.toString(int i, int radix) | returns a string representation of the int type argument in the specified radix.(First int to radix conversion) |
| 12 | Integer Integer.valueOf(int i) | returns the relevant Integer Object holding the value of the argument passed. |
|  | Integer Integer.valueOf(String s) | returns the relevant Integer Object holding the value of the argument passed. |
|  | Integer Integer.valueOf(String s, int radix) | convert to int and chnages be base from radix to 10. |

\*valueOf is present in both Integer and String, Integer.valueOf() give Integer and String.valueOf() gives String

**Collections**

Collection<E> P = new "some type"<E>();

Collection<E> Q = new "some type"<E>();

| **No.** | **Methods** | **Description** |
| --- | --- | --- |
| 1 | boolean P.add(E element) | This method is used to add an object to the collection. |
| 2 | boolean P.addAll(Q); | This method adds all the elements in the given collection to this collection. |
| 3 | P.clear() | used to clear the Collection upon |
| 4 | boolean P.contains(Object element) | returns true if the collection contains the specified element. |
| 5 | boolean P.containsAll(Q) | returns true if the collection contains all of the elements in the given collection. |
| 6 | boolean P.equals(Q) | This method compares the specified object with this collection for equality. |
| 7 | int P.hashCode() | return the hash code value for this collection. |
| 8 | boolean P.isEmpty() | This method returns true if this collection contains no elements. |
| 9 | Collections.max(P) | used to return the maximum element of the given collection, according to the natural ordering of its elements. All elements in the collection must implement the Comparable interface. |
| 10 | E P.remove(Object o) | This method is used to remove the given object from the collection. If there are duplicate values, then this method removes the first occurrence of the object. |
| 11 | E P.removeAll(Q) | This method is used to remove all the objects mentioned in the given collection from the collection. |
| 12 | int P.size() | This method is used to return the number of elements in the collection. |
| 13 | Object[] objects = P.toArray(); | This method is used to return an array containing all of the elements in this collection. toArray() method returns an array of type Object(Object[]). We need to typecast it to Integer before using as Integer objects. |

**Iterator**

Syntax :

Iterator i = Collection.iterator();

| **No.** | **Methods** | **Description** |
| --- | --- | --- |
| 1 | boolean i.hasNext() | returns true if Iterator has more element to iterate. |
| 2 | Object i.next() | returns the next element in the collection until the hasNext()method return true. |
| 3 | void i.remove() | removes the current element in the collection. |

ListItrator

‘ListIterator’ in Java is an Iterator which allows users to traverse Collection in both direction.

Syntax:

ListIterator li = list.listIterator();

| **No.** | **Methods** | **Description** |
| --- | --- | --- |
| 1 | void li.add(Object object) | It inserts object immediately before the element that is returned by the next( ) function. |
| 2 | boolean li.hasNext( ) | returns true if the list has a next element. |
| 3 | boolean li.hasPrevious( ): | returns true if the list has a previous element. |
| 4 | Object li.next( ) | returns the next element of the list. |
| 5 | Object li.previous( ) | returns the previous element of the list. |
| 6 | void li.remove( ) | removes the current element from the list. |

**ArrayList**

Syntax:

ArrayList <E> x = new ArrayList<E>();

| **No.** | **Methods** | **Description** |
| --- | --- | --- |
| 1 | int x.size() | returns length of an ArrayList |
| 2 | void x.add(int index, E element) | used to insert the specified element at the specified position in a list. |
| 3 | boolean x.add(E e) | used to append the specified element at the end of a list. |
| 4 | boolean x.addAll(Collection c) | used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. |
| 5 | boolean x.addAll(int index, Collection c) | used to append all the elements in the specified collection, starting at the specified position of the list. |
| 6 | void x.clear() | used to remove all of the elements from this list. |
| 7 | E x.get(int index) | used to fetch the element from the particular position of the list. |
| 8 | boolean x.isEmpty() | returns true if the list is empty, otherwise false. |
| 9 | Object[] x.toArray() | used to return an array containing all of the elements in this list in the correct order. |
| 10 | Object x.clone() | used to return a shallow copy of an ArrayList. |
| 11 | boolean x.contains(Object o) | returns true if the list contains the specified element |
| 12 | int x.indexOf(Object o) | used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element. |
| 13 | E x.remove(int index) | to remove the element present at the specified position in the list. |
| 14 | boolean x.remove(Object o) | used to remove the first occurrence of the specified element. |
|  | boolean x.removeAll(Collection c) | used to remove all the elements from the list. |
| 15 | void x.removeRange(int fromIndex, int toIndex) | to remove all the elements lies within the given range. |
| 16 | E x.set(int index, E element) | used to replace the specified element in the list, present at the specified position. |
| 17 | void x.sort(Comparator<? super E> c) | used to sort the elements of the list on the basis of specified comparator. |
| 18 | List<E> subList(int fromIndex, int toIndex) | used to fetch all the elements lies within the given range. |

**LinkedList**

Syntax:

LinkedList<E> x =new LinkedList<E>();

E: data type

| **No.** | **Methods** | **Description** |
| --- | --- | --- |
| 1 | boolean x.add(E e) | to append the specified element to the end of a list. |
| 2 | void x.add(int index, E element) | to insert the specified element at the specified position index in a list. |
| 3 | boolean x.addAll(Collection <E> c) | used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. |
|  | boolean x.addAll(int index, Collection<E> c) | used to append all the elements in the specified collection, starting at the specified position of the list. |
| 4 | void x.addFirst(E e) | used to insert the given element at the beginning of a list. |
|  | void x.addLast(E e) | used to append the given element to the end of a list. |
| 5 | void x.clear() | used to remove all the elements from a list. |
| 6 | Object x.clone() | used to return a shallow copy of an ArrayList. |
| 7 | boolean x.contains(Object o) | used to return true if a list contains a specified element. |
| 8 | E x.element() | used to retrieve the first element of a list. |
| 9 | E x.get(int index) | used to return the element at the specified position in a list. |
| 10 | int x.indexOf(Object o) | used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element. |
| 11 | boolean x.offer(E e) | adds the specified element as the last element of a list. |
| 12 | E x.peek() | retrieves the first element of a list |
|  | E x.poll() | It retrieves and removes the first element of a list. |
| 13 | int x.size() | used to return the number of elements in a list. |
| 14 | E x.set(int index, E element) | It replaces the element at the specified position in a list with the specified element. |
| 15 | E x.remove(int index) | used to remove the element at the specified position in a list. |

**Map**

Map<K,V> m=new HashMap<K,V>();

| **No.** | **Methods** | **Description** |
| --- | --- | --- |
| 1 | void m.put(Object key, Object value) | used to insert an entry in the map. |
| 2 | void m.putAll(Map map) | used to insert the specified map in the map. |
| 3 | void m.putIfAbsent(K key, V value) | It inserts the specified value with the specified key in the map only if it is not already specified. |
| 4 | void m.remove(Object key) | used to delete an entry for the specified key. |
| 5 | boolean m.remove(Object key, Object value) | It removes the specified values with the associated specified keys from the map. |
| 6 | void m.clear() | used to reset the map. |
| 7 | boolean m.containsValue(Object value) | This method returns true if some value equal to the value exists within the map, else return false. |
|  | boolean containsKey(Object key) | This method returns true if some key equal to the key exists within the map, else return false. |
| 8 | boolean equals(Object o) | used to compare the specified Object with the Map. |
| 9 | V m.get(Object key) | This method returns the object that contains the value associated with the key else returns null |
| 10 | boolean m.isEmpty() | This method returns true if the map is empty; returns false if it contains at least one key. |
| 11 | int m.size() | This method returns the number of entries in the map. |

**Queue**

Since Queue is an interface, objects cannot be created of the type queue. We always need a class which extends this list in order to create an object.

Syntax:

Queue<E> pQueue = new PriorityQueue<E>();

Queue<E> lQueue = new LinkedList<E>();

Queue<E> pbq = new PriorityBlockingQueue<E>();

| **No.** | **Methods** | **Description** |
| --- | --- | --- |
| 1 | boolean q.add(object) | This method is used to add elements at the tail of queue. More specifically, at the last of linked-list if it is used, or according to the priority in case of priority queue implementation. |
|  | boolean q.offer(object) | This method is used to insert an element in the queue. This method is preferable to add() method since this method does not throws an exception when the capacity of the container is full since it returns false. |
| 2 | Object q.peek() | This method is used to view the head(first in) of queue without removing it. It returns Null if the queue is empty. |
|  | Object q.element() | This method is similar to peek(). It throws NoSuchElementException when the queue is empty. |
| 3 | Object q.remove() | This method removes and returns the head of the queue. It throws NoSuchElementException when the queue is empty. |
|  | Object q.poll() | This method removes and returns the head of the queue. It returns null if the queue is empty. |
|  |  |  |

**1. Map // collection**  
1. Definition ->Map<K, V> map = new HashMap<>();  
2. insert / update -> V put(k1, v1); // TC: O(1)  
3. delete -> V remove(k1); // TC: O(1)  
4. get -> V get(k1); // TC: O(1)  
5. size -> int size(); // TC: O(1)  
6. check for Empty -> boolean isEmpty(); // TC: O(1)  
7. value present -> boolean containsKey(k1); // TC: O(1)  
8. remove all map values -> clear(); // TC: O(2n + 1) -> O(n) *(n-key, n-value, 1 for map itself)*

**2. ArrayList // Collection**  
1. Definition -> ArrayList list = new ArrayList<>();  
2. insert -> boolean add(t) **[TC: O(1)]** / add(int index, T) **[TC: O(n)]**  
3. delete -> T remove(int index); // TC: O(n) as you have to shuffle the elements above that point  
4. set/update index value -> T set(int index, T); // TC: O(1)  
5. get index-> T get(int index); // TC: O(1)  
6. size -> int list.size(); // TC: O(1)  
7. clear elements -> void clear(); // TC: O(n) & removeAll : O(n^2).  
8. check for Empty -> boolean isEmpty(); // TC: O(1)  
9. value contain check -> boolean contains(t); // TC: O(n)  
10. get Index of value -> int indexOf(t); // TC: O(n), checking each element one by one  
11. non premitive to premitive list -> toArray(); // TC: O(n)  
12. Sorting for List ->

* Collections.sort(list, (a, b) -> a - b); // ascending , TC: O(nlogn)
* Collections.sort(list, (a, b) -> b - a); // descnding , TC: O(nlogn)

**3. Array**  
1. Definition ->T arr [ ]= new T[N]; // N: static size , T : datatype  
2. insert -> arr[index] = v1; // TC: O(1)  
3. update -> arr[index] = v2; // TC: O(1)  
4. get -> T arr[index] // TC: O(1)  
5. size -> int arr.length // TC: O(1)  
6. Arrays.fill(arr, 0); // filled array with value=0, TC: O(n)  
7. Sorting -> TC: O(nlogn)

* premitive (int[] ..)
  + Arrays.sort(arr); // default ascending,
* non-premetive (Integer[] ..)
  + Arrays.sort(arr); // default ascending
  + Arrays.sort(arr, (a,b) -> b-a); // descening

**4. Stack // Collection**  
1. Definition ->Stack st = new Stack<>();  
2. insert -> T push(t); // TC: O(1)  
3. size -> int size(); // TC: O(1)  
4. look up for head element -> T peek(); // TC: O(1)  
5. remove head element -> T pop(); // TC: O(1)  
6. check for Empty -> boolean isEmpty(); // TC: O(1)

**5. Queue // Collection**  
1. Definition -> Queue queue = new LinkedList<>();  
2. insert -> boolean add(t); // TC: O(1)  
3. size -> int size(); // TC: O(1)  
4. look up for head element -> T peek(); // TC: O(1)  
5. remove head element -> T poll(); // TC: O(1)  
6. check for Empty -> boolean isEmpty(); // TC: O(1)  
7. points to remember :

* queue poll vs stack pop
* queue add vs stack push
* we can define queue via LinkedList, PriorityQueue based on use case

**6. String / StringBuilder**  
1. Definition -> String str = new String();  
2. size -> int length();// TC: O(1)  
3. convert to char Array -> toCharArray(); // TC: O(n)  
4. value for specific index -> charAt(int index); // TC: O(1)  
5. substring from string -> substring [a,b) // a : inclusive, b: Exclusive, TC: O(n)  
6. transform to Lowercase -> toLowerCase(); // TC: O(n)  
7. transform to UpperCase -> toUpperCase(); // TC: O(n)  
8. replace all characters in string -> replaceAll(from, to) // TC: O(n)  
9. Some useful Character properties

* Character.isLetter();
* Character.isAlphabetic();
* Character.isUpperCase();
* Character.isLowerCase();
* Character.isDigit();

1. Concatenation

* T str1 + str2
* StringBuilder ->
  + new StringBuilder() / new StringBuilder(int)
  + append("adding string") // better way to do
  + toString() // converting back to string

**7. HashSet // Collection**  
1. Definition ->Set set = new HashSet<>();  
2. insert / update -> boolean add(t); // TC: O(1)  
3. delete -> boolean remove(t); // TC: O(1)  
4. get -> boolean contains(t); // TC: O(1)  
5. size -> int size(); // TC: O(1)  
6. check for Empty -> boolean isEmpty(); // TC: O(1)  
7. remove all set values -> clear(); // TC: O(n)

## **Arrays**

* **Set, Check** element at a particular index: **O(1)**
* **Searching**: **O(n)** if array is unsorted and **O(log n)** if array is sorted and something like a binary search is used,
* As pointed out by [Aivean](https://stackoverflow.com/users/1349366/aivean), there is no Delete operation available on Arrays. We can symbolically delete an element by setting it to some specific value, e.g. -1, 0, etc. depending on our requirements
* Similarly, Insert for arrays is basically Set as mentioned in the beginning

## **ArrayList:**

* **Add**: **Amortized O(1)**
* **Remove**: **O(n)**
* **Contains**: **O(n)**
* **Size**: **O(1)**

## **Linked List:**

* **Inserting**: **O(1)**, if done at the head, **O(n)** if anywhere else since we have to reach that position by traversing the linkedlist linearly.
* **Deleting**: **O(1)**, if done at the head, **O(n)** if anywhere else since we have to reach that position by traversing the linkedlist linearly.
* **Searching**: **O(n)**

## **Doubly-Linked List:**

* **Inserting**: **O(1)**, if done at the head or tail, **O(n)** if anywhere else since we have to reach that position by traversing the linkedlist linearly.
* **Deleting**: **O(1)**, if done at the head or tail, **O(n)** if anywhere else since we have to reach that position by traversing the linkedlist linearly.
* **Searching**: **O(n)**

## **Stack:**

* **Push**: **O(1)**
* **Pop**: **O(1)**
* **Top**: **O(1)**
* **Search** (Something like lookup, as a special operation): **O(n)** (I guess so)

## **Queue/Deque/Circular Queue:**

* **Insert**: **O(1)**
* **Remove**: **O(1)**
* **Size**: **O(1)**

## **Binary Search Tree:**

* **Insert, delete and search**: Average case: **O(log n)**, Worst Case: **O(n)**

## **Red-Black Tree:**

* **Insert, delete and search**: Average case: **O(log n)**, Worst Case: **O(log n)**

## **Heap/PriorityQueue (min/max):**

* **Find Min/Find Max**: **O(1)**
* **Insert**: **O(log n)**
* **Delete Min/Delete Max**: **O(log n)**
* **Extract Min/Extract Max**: **O(log n)**
* **Lookup, Delete** (if at all provided): **O(n)**, we will have to scan all the elements as they are not ordered like BST

## **HashMap/Hashtable/HashSet:**

* **Insert/Delete**: **O(1)** amortized
* **Re-size/hash**: **O(n)**
* **Contains**: **O(1)**

<https://www.baeldung.com/java-collections-complexity>

JAVA PASS BY VALUE?REFERENCE

**Call-by-reference or pass-by-reference is not supported in Java.** Java supports **pass-by-value** only. In **pass-by-value**, any changes to a parameter inside the called method are not reflected outside the method's scope.

In pass-by-reference, changes made to the parameters inside the method are also reflected outside. Though Java does not support pass-by-reference, we can achieve pass-by-reference in Java for non-primitives.

 we don’t have pointers in Java.

Basically, a pointer is a variable that stores the memory address of another variable. These refer to the methodology that is used while passing a variable as an argument to a method.

A diagram of a function

Description automatically generated

\*ptr reveals the value of the variable that is present at the memory address stored in ptr. In this case \*ptr = 10;

## Pass by Value in Java

When we pass only the value part of a variable to a function as an argument, it is referred to as **pass by value**. In this case, any change to the value of a parameter in the called method does not affect its value in the calling method.

As can be seen in the figure below, only the value part of the variable is passed i.e. a copy of the existing variable is passed instead of passing the origin variable. Hence, any changes done to the value of the copy will not have any impact on the value of the original variable. Java supports pass-by-value.

When reference to the location of a variable is passed to a function as an argument, then it is called **pass by reference**. In this case, any changes to the value of a parameter inside the function are reflected outside as well.

As we can see in the below figure, the memory address of the variable is stored in a pointer, and this pointer is passed to the function as arguments. In this case, the called function will have access to the original address of the variable. Thus, if we change the value of the variable inside the method, the change is reflected outside the method as well.

Java does not support pass-by-reference as it does not have the concept of pointers. But we can achieve pass-by-reference in Java through the following ways:

## How Java Handles Pass by Reference Using Pass by Value

In Java, when we create a variable of class type, the variable holds the reference to the object in the heap memory. This reference is stored in the stack memory.

Hence, when we pass the variable as an argument to a method, we inherently pass a copy of the reference to the object in the heap memory. As a result, the method parameter that receives the object refers to the same object as that referred to by the argument.

Thus, changes to the **properties of the object** inside the method are reflected outside as well. This effectively means that objects are passed to methods by use of call-by-reference.

**Changes to the properties of an object inside a method affect the original argument as well.** However, if we change the object altogether, then the original object is not changed. Instead a new object is created in the heap memory and that object is assigned to the copied reference variable passed as argument.

### Case with Primitive Data types

This is the case with Java Objects only. Primitive data types are allocated memory in the stack memory, not in the heap memory. Hence, when we pass a variable of primitive data type, a copy of the variable is created in the stack memory, and it is passed as an argument.

As a result, any changes to the copied variable inside the called method are not reflected in the original argument.

**Important:**

Java does not support pass-by-reference or call-by-reference by any means. The above scenario occurs due to the way objects are created in Java. Java object variables are simply references that point to real objects in the memory heap.

Java supports pass-by-value only. However, **copying the reference to an object in the stack memory does not create a clone of the object in the heap memory.** Hence when the copied reference is sent as an argument, changes to the fields of the referenced object are reflected in the original fields.

**Difference between parameters and arguments:**

**Parameters:** Method parameters are the names listed in the method definition.

**Arguments:** Method arguments are the real values passed to the method.

public class PassByValue {

public static void main(String[] args) {

int a = 2;

int b = 3;

add(a, b);

System.out.println("Result from main: " + (a + b));

}

private static void add(int a, int b) {

a = 10;

System.out.println("Result from method: " + (a + b));

}

}

Result from method: 13

Result from main: 5

* When the variables a and b are passed as an argument to the add method, copies of these variables are created in the stack memory.
* These copied variables are passed to the add method.
* Hence, when a is changed inside the add method, it does affect the variable a defined in the main method.

public class PassByValue {

public static void main(String[] args) {

Integer a = new Integer(2);

Integer b = new Integer(3);

add(a, b);

System.out.println("Result from main: " + (a + b));

}

private static void add(Integer a, Integer b) {

// If a is simply assigned a reference to a new object

// in heap memory, the original argument in main method

// does not change

a = new Integer(10);

System.out.println("Result from method: " + (a + b));

}

}

Result from method: 13

Result from main: 5

* In the above program, variables a and b in the main method store the references of both Integer objects created in the heap memory.
* Passing a and b as arguments to the add method creates copies of these variables/references in the stack memory.
* Point to note here is that these copied variables store the references to the same Integer objects created in the main method.
* Inside the add method, we are changing the object altogether. The a variable inside the add method is made to store the reference to a new Integer object created in the heap memory with the value 10.
* However, due to this operation, only the copied reference variable is affected and the original variable in the main method is not affected.
* Since there are no changes made to the Integer objects created in the main method, the sum inside the main method is 5 only.

class PassByValue {

public static void main(String[] args) {

Integer[] array = new Integer[2];

array[0] = 2;

array[1] = 3;

add(array);

System.out.println("Result from main: " + (array[0] + array[1]));

}

private static void add(Integer[] array) {

array[0] = 10;

System.out.println("Result from method: " + (array[0] + array[1]));

}

}

Result from method: 13

Result from main: 13

* **The called method is able to modify the original object but not replace it with another object.**
* In this example, when we create the array object in the main method, a new Integer array object is created in the heap memory. The array variable in the main method holds the reference to the same object.
* On passing array as an argument to the add method creates a copy of that reference in the stack memory. This copied reference/variable is passed to the add method.
* However, the point to note here is that the copied reference points to the same Integer array object that we created in the first step.
* Hence, when we modify the properties of the Integer array object inside the add method using the copied reference, these modifications are reflected in the heap memory.
* If we had reinitialized the array object with array = new Integer[2]; in the add() method, our result would have been different. Let’s see what the result would be.

class PassByValue {

public static void main(String[] args) {

Integer[] array = new Integer[2];

array[0] = 2;

array[1] = 3;

add(array);

System.out.println("Result from main: " + (array[0] + array[1]));

}

private static void add(Integer[] array) {

array = new Integer[2];

array[0] = 10;

array[1] = 3;

System.out.println("Result from method: " + (array[0] + array[1]));

}

}

Result from method: 13

Result from main: 5

**Explanation:**

* Here, we can see that if we reinitialize the array object, which is passed in arguments, the original reference breaks(i.e., we have replaced the original object reference with some other object reference), and the array no longer is referenced to the original array. Hence the value in the main() method didn’t change.
* **Important Note:**
* The modification of an object depends on the immutability of a Java class. The classes like Integer, String, Float, Double, Byte, Long, Short, Boolean, and Character are all immutable classes; hence, once created, no modification can be made on the same reference. Hence these classes will strictly follow pass-by-value methodology, same as primitive data types.
* Java supports pass-by-value only.
* Java doesn’t support pass-by-reference.
* Primitive data types and Immutable class objects strictly follow pass-by-value; hence can be safely passed to functions without any risk of modification.
* For non-primitive data types, Java sends a copy of the reference to the objects created in the heap memory.
* Any modification made to the referenced object inside a method will reflect changes in the original object.
* If the referenced object is replaced by any other object, any modification made further will not impact the original object.

# How to create Immutable class in Java?

 [Read](javascript:void(0))

 [Discuss(20+)](javascript:void(0))

 [Courses](javascript:void(0))

 [Practice](javascript:void(0))

Immutable class in java means that once an object is created, we cannot change its content. In Java, all the [wrapper classes](https://www.geeksforgeeks.org/wrapper-classes-java/) (like Integer, Boolean, Byte, Short) and String class is immutable. We can create our own immutable class as well. Prior to going ahead do go through characteristics of immutability in order to have a good understanding while implementing the same. Following are the requirements:

* The class must be declared as final so that child classes can’t be created.
* Data members in the class must be declared private so that direct access is not allowed.
* Data members in the class must be declared as final so that we can’t change the value of it after object creation.
* A parameterized constructor should initialize all the fields performing a deep copy so that data members can’t be modified with an object reference.
* Deep Copy of objects should be performed in the getter methods to return a copy rather than returning the actual object reference)

***Note:****There should be no setters or in simpler terms, there should be no option to change the value of the instance variable.*

**Example**

* Java

|  |
| --- |
| // Java Program to Create An Immutable Class    // Importing required classes  **import** java.util.HashMap;  **import** java.util.Map;    // Class 1  // An immutable class  **final** **class** Student {        // Member attributes of final class  **private** **final** String name;  **private** **final** **int** regNo;  **private** **final** Map<String, String> metadata;        // Constructor of immutable class      // Parameterized constructor  **public** Student(String name, **int** regNo,                     Map<String, String> metadata)      {            // This keyword refers to current instance itself  **this**.name = name;  **this**.regNo = regNo;            // Creating Map object with reference to HashMap          // Declaring object of string type          Map<String, String> tempMap = **new** HashMap<>();            // Iterating using for-each loop  **for** (Map.Entry<String, String> entry :               metadata.entrySet()) {              tempMap.put(entry.getKey(), entry.getValue());          }    **this**.metadata = tempMap;      }        // Method 1  **public** String getName() { **return** name; }        // Method 2  **public** **int** getRegNo() { **return** regNo; }        // Note that there should not be any setters        // Method 3      // User -defined type      // To get meta data  **public** Map<String, String> getMetadata()      {            // Creating Map with HashMap reference          Map<String, String> tempMap = **new** HashMap<>();    **for** (Map.Entry<String, String> entry :  **this**.metadata.entrySet()) {              tempMap.put(entry.getKey(), entry.getValue());          }  **return** tempMap;      }  }    // Class 2  // Main class  **class** GFG {        // Main driver method  **public** **static** **void** main(String[] args)      {            // Creating Map object with reference to HashMap          Map<String, String> map = **new** HashMap<>();            // Adding elements to Map object          // using put() method          map.put("1", "first");          map.put("2", "second");            Student s = **new** Student("ABC", 101, map);            // Calling the above methods 1,2,3 of class1          // inside main() method in class2 and          // executing the print statement over them          System.out.println(s.getName());          System.out.println(s.getRegNo());          System.out.println(s.getMetadata());            // Uncommenting below line causes error          // s.regNo = 102;            map.put("3", "third");          // Remains unchanged due to deep copy in constructor          System.out.println(s.getMetadata());          s.getMetadata().put("4", "fourth");          // Remains unchanged due to deep copy in getter          System.out.println(s.getMetadata());      }  } |

**Output**

ABC

101

{1=first, 2=second}

{1=first, 2=second}

{1=first, 2=second}

In this example, we have created a final class named Student. It has three final data members, a parameterized constructor, and getter methods. Please note that there is no setter method here. Also, note that we don’t need to perform deep copy or cloning of data members of wrapper types as they are already immutable.

### What are Immutable objects?

Immutable objects are objects which once declared elements can’t be modified after it.

et us perform some more functions:

// assigns a new reference to the

// same string "knowledge"

String s = str;

**Let’s see how the below statement works:**

str = str.concat(" base");

This appends a string ” base” to str. But wait, how is this possible, since String objects are immutable? Well to your surprise, it is.

When the above statement is executed, the VM takes the value of String str, i.e. “knowledge” and appends ” base”, giving us the value “knowledge base”. Now, since Strings are immutable, the VM can’t assign this value to str, so it creates a new String object, gives it a value “knowledge base”, and gives it reference str.

An important point to note here is that, while the String object is immutable, **its reference variable is not.** So that’s why, in the above example, the reference was made to refer to a newly formed String object.

At this point in the example above, we have two String objects: the first one we created with the value “knowledge”, pointed to by s, and the second one “knowledge base”, pointed to by str. But, technically, we have three String objects, the third one being the literal “base” in the concat statement.

### Why Java Strings are immutable in nature?

These are some more reasons for making String immutable in Java. These are:

* The String pool cannot be possible if String is not immutable in Java. A lot of heap space is saved by JRE. The same string variable can be referred to by more than one string variable in the pool. String interning can also not be possible if the String would not be immutable.
* If we don’t make the String immutable, it will pose a serious security threat to the application. For example, database usernames, and passwords are passed as strings to receive database connections. The socket programming host and port descriptions are also passed as strings. The String is immutable, so its value cannot be changed. If the String doesn’t remain immutable, any hacker can cause a security issue in the application by changing the reference value.
* The String is safe for multithreading because of its immutableness. Different threads can access a single “String instance”. It removes the synchronization for thread safety because we make strings thread-safe implicitly.
* Immutability gives the security of loading the correct class by Classloader. For example, suppose we have an instance where we try to load java.sql.Connection class but the changes in the referenced value to the myhacked.The connection class does unwanted things to our database.

### Important Facts about String and Memory Usage

What if we didn’t have another reference s to “knowledge”? We would have lost that String. However, it still would have existed but would be considered lost due to having no references.   
Look at one more example below

### Example

The below programs demonstrate the immutability of Java strings.

* Java

|  |
| --- |
| // Java Program to demonstrate why  // Java Strings are immutable    **import** java.io.\*;    **class** GFG {  **public** **static** **void** main(String[] args)      {          String s1 = "java";          s1.concat(" rules");            // Yes, s1 still refers to "java"          System.out.println("s1 refers to " + s1);      }  } |

**Output**

s1 refers to java

#### **Explanation of the above method**

1. The first line is pretty straightforward: create a new String “java” and refer s1 to it.
2. Next, the VM creates another new String “java rules”, but nothing refers to it. So, the second String is instantly lost. We can’t reach it.

The reference variable s1 still refers to the original string “java”.

## Frequently Asked Questions (FAQs)

### Q1: Where do these String objects go?

**Answer:**

*Well, these exist in memory, and one of the key goals of any programming language is to make efficient use of memory.As applications grow, it’s very common for String literals to occupy a large area of memory, which can even cause redundancy. So, in order to make Java more efficient,****the JVM sets aside a special area of memory called the “String constant pool”.****When the compiler sees a String literal, it looks for the String in the pool. If a match is found, the reference to the new literal is directed to the existing String and no new String object is created. The existing String simply has one more reference. Here comes the point of making String objects immutable:*

*In the String constant pool, a String object is likely to have one or many references. If several references point to the same String without even knowing it, it would be bad if one of the references modified that String value. That’s why String objects are immutable.*

### Q2: What happens if someone overrides the functionality of the String class?

**Answer:**

***As String class is marked final****so that nobody can override the behavior of its methods.*

## **4. Shallow Copy**

A shallow copy is one in which **we only copy values of fields** from one object to another:

@Test

**public** **void** **whenShallowCopying\_thenObjectsShouldNotBeSame**() {

**Address** address = **new** **Address**("Downing St 10", "London", "England");

**User** pm = **new** **User**("Prime", "Minister", address);

**User** shallowCopy = **new** **User**(

pm.getFirstName(), pm.getLastName(), pm.getAddress());

assertThat(shallowCopy)

.isNotSameAs(pm);

}Copy

In this case, pm != shallowCopy, which means that **they're different objects; however, the problem is that when we change any of the original address' properties, this will also affect the shallowCopy‘s address**.

## **5. Deep Copy**

A deep copy is an alternative that solves this problem. Its advantage is that **each mutable object in the object graph is recursively copied**.

Since the copy isn't dependent on any mutable object that was created earlier, it won't get modified by accident like we saw with the shallow copy.

In the following sections, we'll discuss several deep copy implementations and demonstrate this advantage.

### ****5.1. Copy Constructor****

The first implementation we'll examine is based on copy constructors:

**public** **Address**(Address that) {

this(that.getStreet(), that.getCity(), that.getCountry());

}Copy

**public** **User**(User that) {

this(that.getFirstName(), that.getLastName(), **new** **Address**(that.getAddress()));

}Copy

In the above implementation of the deep copy, we haven't created new Strings in our copy constructor because String is an immutable class.

As a result, they can't be modified by accident. Let's see if this works:

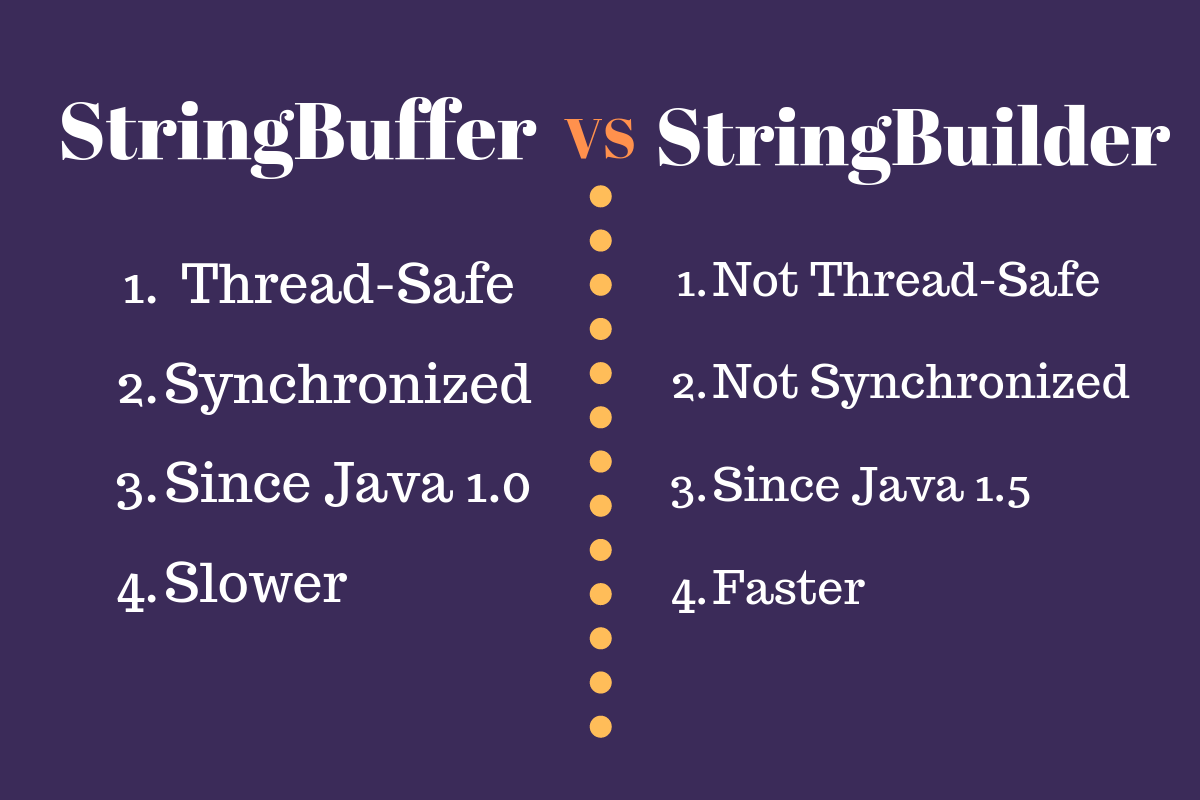
# String vs StringBuffer vs StringBuilder

Published on August 3, 2022

* [Java](https://www.digitalocean.com/community/tags/java)
* [String](https://www.digitalocean.com/community/tags/string)



By Pankaj



While we believe that this content benefits our community, we have not yet thoroughly reviewed it. If you have any suggestions for improvements, please let us know by clicking the “report an issue“ button at the bottom of the tutorial.

String is one of the most widely used classes in Java. StringBuffer and StringBuilder classes provide methods to manipulate strings. We will look into the difference between StringBuffer and StringBuilder. StringBuffer vs StringBuilder is a popular Java interview question.

## [String vs StringBuffer vs StringBuilder](https://www.digitalocean.com/community/tutorials/string-vs-stringbuffer-vs-stringbuilder#string-vs-stringbuffer-vs-stringbuilder)

The string is one of the most important topics in the core java interview. If you are writing a program that prints something on the console, you are use String. This tutorial is aimed to focus on major features of String class. Then we will compare the StringBuffer and StringBuilder classes.

## [String in Java](https://www.digitalocean.com/community/tutorials/string-vs-stringbuffer-vs-stringbuilder#string-in-java)

1. String class represents character strings, we can instantiate String in two ways.
2. String str = "ABC";
3. // or
4. String str = new String("ABC");
5. String is immutable in Java. So it’s suitable to use in a multi-threaded environment. We can share it across functions because there is no concern of data inconsistency.
6. When we create a String using double quotes, JVM first looks for the String with the same value in the string pool. If found, it returns the reference of the string object from the pool. Otherwise, it creates the String object in the String pool and returns the reference. JVM saves a lot of memory by using the same String in different threads.
7. If the new operator is used to create a string, it gets created in the heap memory.
8. The + operator is overloaded for String. We can use it to concatenate two strings. Although internally it uses StringBuffer to perform this action.
9. String overrides [equals() and hashCode()](https://www.digitalocean.com/community/tutorials/java-equals-hashcode) methods. Two Strings are equal only if they have the same character sequence. The equals() method is case sensitive. If you are looking for case insensitive checks, you should use equalsIgnoreCase() method.
10. The string uses UTF-16 encoding for the character stream.
11. String is a final class. All the fields as final except “private int hash”. This field contains the hashCode() function value. The hashcode value is calculated only when the hashCode() method is called for the first time and then cached in this field. Furthermore, the hash is generated using the final fields of String class with some calculations. So every time the hashCode() method is called, it will result in the same output. For the caller, it seems like calculations are happening every time but internally it’s cached in the hash field.

## [String vs StringBuffer](https://www.digitalocean.com/community/tutorials/string-vs-stringbuffer-vs-stringbuilder#string-vs-stringbuffer)

Since String is immutable in Java, whenever we do String manipulation like concatenation, substring, etc. it generates a new String and discards the older String for garbage collection. These are heavy operations and generate a lot of garbage in heap. So Java has provided StringBuffer and StringBuilder classes that should be used for String manipulation. StringBuffer and StringBuilder are mutable objects in Java. They provide append(), insert(), delete(), and substring() methods for String manipulation.

## [StringBuffer vs StringBuilder](https://www.digitalocean.com/community/tutorials/string-vs-stringbuffer-vs-stringbuilder#stringbuffer-vs-stringbuilder)

StringBuffer was the only choice for String manipulation until Java 1.4. But, it has one disadvantage that all of its public methods are synchronized. StringBuffer provides Thread safety but at a performance cost. In most of the scenarios, we don’t use String in a multithreaded environment. So Java 1.5 introduced a new class StringBuilder, which is similar to StringBuffer except for thread-safety and synchronization. StringBuffer has some extra methods such as substring, length, capacity, trimToSize, etc. However, these are not required since you have all these present in String too. That’s why these methods were never implemented in the StringBuilder class. StringBuffer was introduced in Java 1.0 whereas StringBuilder class was introduced in Java 1.5 after looking at shortcomings of StringBuffer. If you are in a single-threaded environment or don’t care about thread safety, you should use StringBuilder. Otherwise, use StringBuffer for thread-safe operations.

## [StringBuilder vs StringBuffer Performance](https://www.digitalocean.com/community/tutorials/string-vs-stringbuffer-vs-stringbuilder#stringbuilder-vs-stringbuffer-performance)

I am trying to check the effect on performance because of synchronization with a sample program that performs append() on StringBuffer and StringBuilder object for multiple times.

package com.journaldev.java;

import java.util.GregorianCalendar;

public class TestString {

public static void main(String[] args) {

System.gc();

long start=new GregorianCalendar().getTimeInMillis();

long startMemory=Runtime.getRuntime().freeMemory();

StringBuffer sb = new StringBuffer();

//StringBuilder sb = new StringBuilder();

for(int i = 0; i<10000000; i++){

sb.append(":").append(i);

}

long end=new GregorianCalendar().getTimeInMillis();

long endMemory=Runtime.getRuntime().freeMemory();

System.out.println("Time Taken:"+(end-start));

System.out.println("Memory used:"+(startMemory-endMemory));

}

}

I ran the same code for the StringBuffer object also to check the time and memory values. I have executed the code 5 times for each case and then calculated the average values.

| **Value of i** | **StringBuffer (Time, Memory)** | **StringBuilder (Time, Memory)** |
| --- | --- | --- |
| 10,00,000 | 808, 149356704 | 633, 149356704 |
| 1,00,00,000 | 7448, 147783888 | 6179, 147783888 |

It’s clear that StringBuilder performs better than StringBuffer even in the case of a single-threaded environment. This difference in performance can be caused by synchronization in StringBuffer methods.

## [String vs StringBuffer vs StringBuilder](https://www.digitalocean.com/community/tutorials/string-vs-stringbuffer-vs-stringbuilder#string-vs-stringbuffer-vs-stringbuilder)

1. String is immutable whereas StringBuffer and StringBuilder are mutable classes.
2. StringBuffer is thread-safe and synchronized whereas StringBuilder is not. That’s why StringBuilder is faster than StringBuffer.
3. String concatenation operator (+) internally uses StringBuffer or StringBuilder class.
4. For String manipulations in a non-multi threaded environment, we should use StringBuilder else use StringBuffer class.

That’s all for a quick roundup of difference be

In Java, an immutable class is a class (Integer, Byte, Long, Float, Double, Character, Boolean, and Short) which once created then its body can not be changed and the same applies to immutable objects which once created cannot be changed. Now the question arises that we do need the help of wrapper classes while dealing with primitive data types in Java where these classes are made immutable.

Geeks, now you must be wondering why do we make them as immutable for which we will be listing some advantages as listed below as follows:

* They are automatically synchronized as their state can not be altered by virtue of the definition of immutability.
* There are zero scopes of inconsistency as an object of the wrapper class can not be altered.
* It helps in caching as one instance of a specific type itself can facilitate dozen of applications.

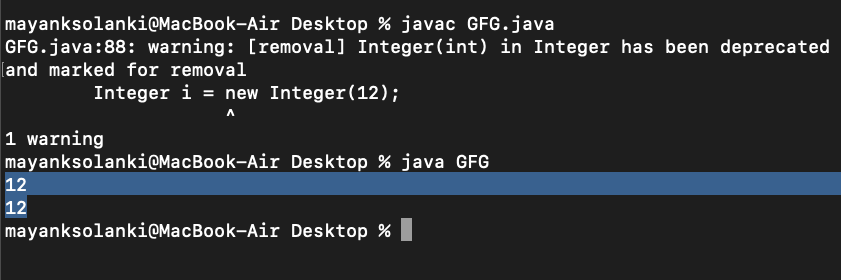
***Tip:****Best usage of wrapper classes as immutable objects is as keys of the Map.*

**Example:**

* Java

|  |
| --- |
| // Java Program to Demonstrate that Primitive  // Wrapper Classes are Immutable    // Main class  **class** GFG {        // Main driver method  **public** **static** **void** main(String[] args)      {            // Getting an integer value          Integer i = **new** Integer(12);            // Printing the same integer value          System.out.println(i);            // Calling method 2          modify(i);            // Now printing the value stored in above integer          System.out.println(i);      }        // Method 2      // To modify integer value  **private** **static** **void** modify(Integer i) { i = i + 1; }  } |

**Output:**



**Output explanation:**

Here the parameter ‘i’ is the reference in modifying and refers to the same object as ‘i’ in main(), but changes made to ‘i’ are not reflected in main() method. It is because all primitive wrapper classes (Integer, Byte, Long, Float, Double, Character, Boolean, and Short) are immutable in Java, so operations like addition and subtraction create a new object and not modify the old.   
The below line of code in the modify method is operating on wrapper class Integer, not an int, and does the following as described below as follows:

i = i + 1;

* Unbox ‘i’ to an int value
* Add 1 to that value
* Box the result into another Integer object
* Assign the resulting Integer to ‘i’ (thus changing what object ‘i’ references)

Since object references are passed by value, the action taken in the modified method does not change i that was used as an argument in the call to modify. Thus the main routine still prints 12 after the method returns.

Wrapper class is a predefined class in java and wrapper class objects wrap primitive data type. In simple words, wrapper class converts the datatype to objects.

Primitive datatype are variables therefore it does not belong to any class and cannot be instantiated.This is the reason wrapper class came into picture.

Each primitive datatype is mapped with dedicated wrapper class and below is the list of same -

A table with different types of wrapper class

Description automatically generated

Primitive data type mapped with dedicated Wrapper class.

If you see above image carefully, you will scrutinize primitive data type starts with lower case and wrapper class name begins with upper case. Variable name always begins with lower case and class name begins with upper case.

All wrapper class name are exactly same as data type except char-Character and int-Integer.

Wrapper classes belong to java.lang package.

**Why do we need wrapper class?**

If you have worked with collections like Arraylist, Vectors , HashMap they have generic<> which accepts object not primitive data type. Lets take an example -

ArrayList<Integer> list= new ArrayList<Integer>();

<> will not accept primitive data type, it accepts object. It converts the primitive data-type into objects.

Wrapper class are immutable as object reference are passed by value.