| **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.