# Sirf khwabh Nhi

**29/08**

## 1]Java instanceof Operator

**Java instanceof operator in detail with the help of examples.**

**The `instanceof` operator in Java is used to check whether an object is an instance of a particular class or not.**

**Its syntax is**

```
objectName instanceOf className;
```

**Here, if `objectName` is an instance of `className`, the operator returns `true`. Otherwise, it returns `false`.**

---

**Example: Java instanceof**

```
class Main {


  public static void main(String[] args) {


    // create a variable of string type
    String name = "Programiz";


    // checks if name is instance of String
    boolean result1 = name instanceof String;
    System.out.println("name is an instance of String: " + result1);
```

```
    // create an object of Main
    Main obj = new Main();


    // checks if obj is an instance of Main
    boolean result2 = obj instanceof Main;
    System.out.println("obj is an instance of Main: " + result2);
  }
}
```

**Output**

```
name is an instance of String: true
obj is an instance of Main: true
```

**In the above example, we have created a variable `name` of the `String` type and an object `obj` of the `Main` class.**

**Here, we have used the `instanceof` operator to check whether `name` and `obj` are instances of the `String` and `Main` class respectively. And, the operator returns `true` in both cases.**

> **Note: In Java, `String` is a class rather than a primitive data type.**

---

**Java instanceof during Inheritance**

**We can use the `instanceof` operator to check if objects of the subclass is also an instance of the superclass. For example,**

```
// Java Program to check if an object of the subclass
// is also an instance of the superclass
```

```
// superclass
class Animal {
}

// subclass
class Dog extends Animal {
}

class Main {
  public static void main(String[] args) {

    // create an object of the subclass
    Dog d1 = new Dog();

    // checks if d1 is an instance of the subclass
    System.out.println(d1 instanceof Dog);         // prints true

    // checks if d1 is an instance of the superclass
    System.out.println(d1 instanceof Animal);      // prints true
  }
}
```

**In the above example, we have created a subclass `Dog` that inherits from the superclass `Animal`. We have created an object `d1` of the `Dog` class.**

**Inside the print statement, notice the expression,**

```
d1 instanceof Animal
```

**Here, we are using the `instanceof` operator to check whether `d1` is also an instance of the superclass `Animal`.**

---

**Java instanceof in Interface**

**The `instanceof` operator is also used to check whether an object of a class is also an instance of the interface implemented by the class. For example,**

```
// Java program to check if an object of a class is also
//  an instance of the interface implemented by the class


interface Animal {
}


class Dog implements Animal {
}


class Main {
  public static void main(String[] args) {

    // create an object of the Dog class
    Dog d1 = new Dog();

    // checks if the object of Dog
```

```
    // is also an instance of Animal
    System.out.println(d1 instanceof Animal);  // returns true
  }
}
```

**In the above example, the `Dog` class implements the `Animal` interface. Inside the print statement, notice the expression,**

```
d1 instanceof Animal
```

**Here, `d1` is an instance of `Dog` class. The `instanceof` operator checks if `d1` is also an instance of the interface `Animal`.**

**Note: In Java, all the classes are inherited from the `Object` class. So, instances of all the classes are also an instance of the `Object` class.**

**In the previous example, if we check,**

```
d1 instanceof Object
```

**The result will be `true`.**

## 2]Java this Keyword

we will learn about this keyword in Java, how and where to use them with the help of examples.

### this Keyword

In Java, this keyword is used to refer to the current object inside a method or a constructor. For example,

```
class Main {
    int instVar;
```

```
    Main(int instVar){
        this.instVar = instVar;
        System.out.println("this reference = " + this);
    }

    public static void main(String[] args) {
        Main obj = new Main(8);
        System.out.println("object reference = " + obj);
    }
}
```

Output:

```
this reference = Main@23fc625e
object reference = Main@23fc625e
```

In the above example, we created an object named `obj` of the class `Main`. We then print the reference to the object `obj` and `this` keyword of the class.

Here, we can see that the reference of both `obj` and `this` is the same. It means this is nothing but the reference to the current object.

---

**Use of this Keyword**

There are various situations where `this` keyword is commonly used.

**Using this for Ambiguity Variable Names**

In Java, it is not allowed to declare two or more variables having the same name inside a scope (class scope or method scope). However, instance variables and parameters may have the same name. For example,

```
class MyClass {
```

```
    // instance variable
    int age;


    // parameter
    MyClass(int age){
        age = age;
    }
}
```

In the above program, the instance variable and the parameter have the same name: age. Here, the Java compiler is confused due to name ambiguity.

In such a situation, we use this keyword. For example,

First, let's see an example without using `this` keyword:

```
class Main {


    int age;
    Main(int age){
        age = age;
    }


    public static void main(String[] args) {
        Main obj = new Main(8);
        System.out.println("obj.age = " + obj.age);
    }
```

```
}
```

Output:

```
obj.age = 0
```

In the above example, we have passed `8` as a value to the constructor. However, we are getting `0` as an output. This is because the Java compiler gets confused because of the ambiguity in names between instance the variable and the parameter.

Now, let's rewrite the above code using `this` keyword.

```
class Main {



    int age;

    Main(int age){

        this.age = age;

    }



    public static void main(String[] args) {

        Main obj = new Main(8);

        System.out.println("obj.age = " + obj.age);

    }

}
```

Output:

```
obj.age = 8
```

Now, we are getting the expected output. It is because when the constructor is called, `this` inside the constructor is replaced by the object `obj` that has called the constructor. Hence the `age` variable is assigned value `8`.

Also, if the name of the parameter and instance variable is different, the compiler automatically appends this keyword. For example, the code:

```
class Main {

    int age;



    Main(int i) {

        age = i;

    }

}
```

is equivalent to:

```
class Main {

    int age;



    Main(int i) {

        this.age = i;

    }

}
```

---

**this with Getters and Setters**

Another common use of `this` keyword is in *setters and getters methods* of a class. For example:

```
class Main {

   String name;



   // setter method
```

```
    void setName( String name ) {
        this.name = name;
    }

    // getter method
    String getName(){
        return this.name;
    }

    public static void main( String[] args ) {
        Main obj = new Main();

        // calling the setter and the getter method
        obj.setName("Toshiba");
        System.out.println("obj.name: "+obj.getName());
    }
}
```

Output:

```
obj.name: Toshiba
```

Here, we have used `this` keyword:

- to assign value inside the setter method
- to access value inside the getter method

---

### Using this in Constructor Overloading

While working with constructor overloading, we might have to invoke one constructor from another constructor. In such a case, we cannot call the constructor explicitly. Instead, we have to use `this` keyword.

Here, we use a different form of this keyword. That is, `this()`. Let's take an example,

```
class Complex {


    private int a, b;


    // constructor with 2 parameters
    private Complex( int i, int j ){
        this.a = i;
        this.b = j;
    }


    // constructor with single parameter
    private Complex(int i){
        // invokes the constructor with 2 parameters
        this(i, i);
    }


    // constructor with no parameter
    private Complex(){
```

```
        // invokes the constructor with single parameter
        this(0);
    }

    @Override
    public String toString(){
        return this.a + " + " + this.b + "i";
    }

    public static void main( String[] args ) {

        // creating object of Complex class
        // calls the constructor with 2 parameters
        Complex c1 = new Complex(2, 3);

        // calls the constructor with a single parameter
        Complex c2 = new Complex(3);

        // calls the constructor with no parameters
        Complex c3 = new Complex();

        // print objects
        System.out.println(c1);
```

```
        System.out.println(c2);

        System.out.println(c3);

    }

}
```

Output:

```
2 + 3i

3 + 3i

0 + 0i
```

In the above example, we have used `this` keyword,

- to call the constructor `Complex(int i, int j)` from the constructor `Complex(int i)`
- to call the constructor `Complex(int i)` from the constructor `Complex()`

Notice the line,

```
System.out.println(c1);
```

Here, when we print the object `c1`, the object is converted into a string. In this process, the `toString()` is called. Since we override the `toString()` method inside our class, we get the output according to that method.

One of the huge advantages of `this()` is to reduce the amount of duplicate code. However, we should be always careful while using `this()`.

This is because calling constructor from another constructor adds overhead and it is a slow process. Another huge advantage of using `this()` is to reduce the amount of duplicate code.

> Note: Invoking one constructor from another constructor is called explicit constructor invocation.

---

### Passing this as an Argument

We can use `this` keyword to pass the current object as an argument to a method. For example,

```
class ThisExample {

    // declare variables
    int x;
    int y;


    ThisExample(int x, int y) {
        // assign values of variables inside constructor
        this.x = x;
        this.y = y;

        // value of x and y before calling add()
        System.out.println("Before passing this to addTwo() method:");
        System.out.println("x = " + this.x + ", y = " + this.y);

        // call the add() method passing this as argument
        add(this);

        // value of x and y after calling add()
        System.out.println("After passing this to addTwo() method:");
        System.out.println("x = " + this.x + ", y = " + this.y);
    }
```

```
    void add(ThisExample o){
        o.x += 2;
        o.y += 2;
    }
}

class Main {
    public static void main( String[] args ) {
        ThisExample obj = new ThisExample(1, -2);
    }
}
```

Output:

```
Before passing this to addTwo() method:
x = 1, y = -2
After passing this to addTwo() method:
x = 3, y = 0
```

In the above example, inside the constructor `ThisExample()`, notice the line,

```
add(this);
```

Here, we are calling the `add()` method by passing this as an argument. Since this keyword contains the reference to the object `obj` of the class, we can change the value of `x` and `y` inside the `add()` method.

## 3]Java final keyword

we will learn about Java final variables, methods and classes with examples.

In Java, the `final` keyword is used to denote constants. It can be used with variables, methods, and classes.

Once any entity (variable, method or class) is declared `final`, it can be assigned only once. That is,

- the final variable cannot be reinitialized with another value
- the final method cannot be overridden
- the final class cannot be extended

---

### 1. Java final Variable

In Java, we cannot change the value of a final variable. For example,

```
class Main {

  public static void main(String[] args) {



    // create a final variable

    final int AGE = 32;



    // try to change the final variable

    AGE = 45;

    System.out.println("Age: " + AGE);

  }

}
```

In the above program, we have created a final variable named `age`. And we have tried to change the value of the final variable.

When we run the program, we will get a compilation error with the following message.

```
cannot assign a value to final variable AGE
    AGE = 45;
    ^
```

> Note: It is recommended to use uppercase to declare final variables in Java.

---

**2. Java final Method**

Before you learn about final methods and final classes, make sure you know about the Java Inheritance.

In Java, the `final` method cannot be overridden by the child class. For example,

```
class FinalDemo {
    // create a final method
    public final void display() {
        System.out.println("This is a final method.");
    }
}

class Main extends FinalDemo {
  // try to override final method
  public final void display() {
```

```
    System.out.println("The final method is overridden.");
  }

  public static void main(String[] args) {
    Main obj = new Main();
    obj.display();
  }
}
```

In the above example, we have created a final method named `display()` inside the `FinalDemo` class. Here, the `Main` class inherits the `FinalDemo` class.

We have tried to override the final method in the `Main` class. When we run the program, we will get a compilation error with the following message.

```
display() in Main cannot override display() in FinalDemo
  public final void display() {
                    ^
  overridden method is final
```

---

### 3. Java final Class

In Java, the final class cannot be inherited by another class. For example,

```
// create a final class
final class FinalClass {
  public void display() {
```

```
    System.out.println("This is a final method.");
  }
}


// try to extend the final class
class Main extends FinalClass {
  public  void display() {
    System.out.println("The final method is overridden.");
  }

  public static void main(String[] args) {
    Main obj = new Main();
    obj.display();
  }
}
```

In the above example, we have created a final class named `FinalClass`. Here, we have tried to inherit the final class by the `Main` class.

When we run the program, we will get a compilation error with the following message.

```
cannot inherit from final FinalClass
class Main extends FinalClass {
                   ^
```

**IMP KNOWLEDGE**

## Bubble Sort

The two algorithms that most beginners start their sorting career with would be bubble sort and selection sort. These sorting algorithms are not very efficient, but they provide a key insight into what sorting is and how a sorting algorithm works behind the scenes. Bubble sort relies on multiple swaps instead of a single like selection sort. The algorithm continues to go through the array repeatedly, swapping elements that are not in their correct location.

Algorithm:

1. START
2. Run two loops – an inner loop and an outer loop.
3. Repeat steps till the outer loop are exhausted.
4. If the current element in the inner loop is smaller than its next element, swap the values of the two elements.
5. END

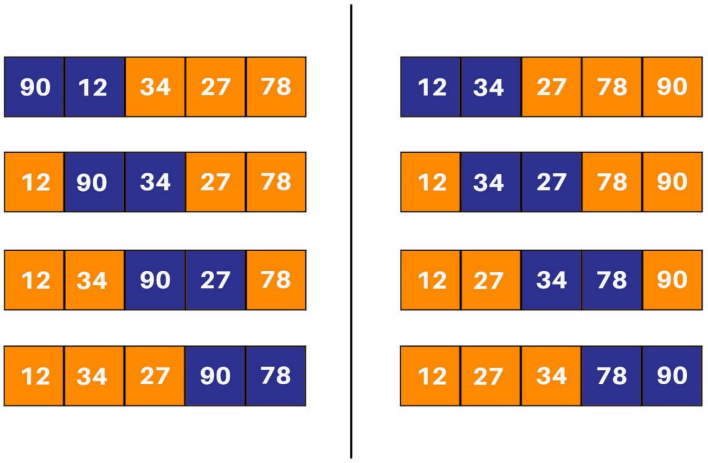
### Bubble Sort Java Code:

```
class Sort

{

    static void bubbleSort(int arr[], int n)

    {

        if (n == 1)                         //passes are done

        {

            return;

        }



        for (int i=0; i<n-1; i++)       //iteration through unsorted
elements
```

```
        {
            if (arr[i] > arr[i+1])      //check if the elements are in order
            {                           //if not, swap them
                int temp = arr[i];
                arr[i] = arr[i+1];
                arr[i+1] = temp;
            }
        }

        bubbleSort(arr, n-1);           //one pass done, proceed to the next
    }

    void display(int arr[])                 //display the array
    {
        for (int i=0; i<arr.length; ++i)
        {
            System.out.print(arr[i]+" ");
        }
    }

    public static void main(String[] args)
    {
        Sort ob = new Sort();
```

```
        int arr[] = {6, 4, 5, 12, 2, 11, 9};

        bubbleSort(arr, arr.length);

        ob.display(arr);

    }

}
```

Explanation of how it works:




**REVISE kaar heee**

## Insertion Sort

If you're quite done with more complex sorting algorithms and want to move on to something simpler: insertion sort is the way to go. While it isn't a much-optimized algorithm for sorting an array, it is one of the more easily understood ones. Implementation is pretty easy too. In insertion sort, one picks up an element and considers it to be the key. If the key is smaller than its predecessor, it is shifted to its correct location in the array.

Algorithm:

1. START
2. Repeat steps 2 to 4 till the array end is reached.

3. Compare the element at current index i with its predecessor. If it is smaller, repeat step 3.
4. Keep shifting elements from the “sorted” section of the array till the correct location of the key is found.
5. Increment loop variable.
6. END

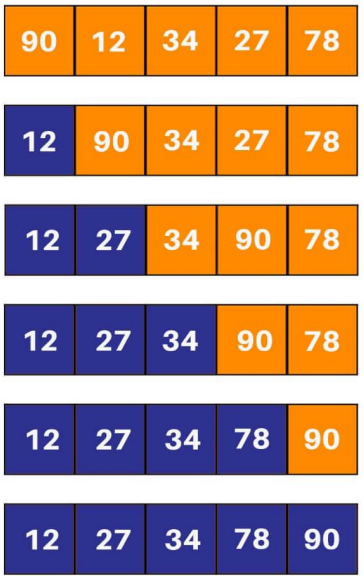
Insertion Sort Java Code:

```
class Sort

{

    static void insertionSort(int arr[], int n)

    {

        if (n <= 1)                                  //passes are done

        {

            return;

        }



        insertionSort( arr, n-1 );          //one element sorted, sort the
remaining array



        int last = arr[n-1];                          //last element of the
array

        int j = n-2;                                  //correct index of last
element of the array



        while (j >= 0 && arr[j] > last)                  //find the correct
index of the last element
```

```
        {
            arr[j+1] = arr[j];                          //shift section of
sorted elements upwards by one element if correct index isn't found
            j--;
        }
        arr[j+1] = last;                            //set the last element at
its correct index
    }

    void display(int arr[])                 //display the array
    {
        for (int i=0; i<arr.length; ++i)
        {
            System.out.print(arr[i]+" ");
        }
    }

    public static void main(String[] args)
    {
        int arr[] = {22, 21, 11, 15, 16};

        insertionSort(arr, arr.length);
        Sort ob = new Sort();
```

```
        ob.display(arr);

    }

}
```

Explanation of how it works:




## Selection Sort

Quadratic sorting algorithms are some of the more popular sorting algorithms that are easy to understand and implement. These don't offer a unique or optimized approach for sorting the array - rather they should offer building blocks for the concept of sorting itself for someone new to it. In selection sort, two loops are used. The inner loop one picks the minimum element from the array and shifts it to its correct index indicated by the outer loop. In every run of the outer loop, one element is shifted to its correct location in the array. It is a very popular sorting algorithm in python as well.

Algorithm:

1. START
2. Run two loops: an inner loop and an outer loop.
3. Repeat steps till the minimum element are found.
4. Mark the element marked by the outer loop variable as a minimum.

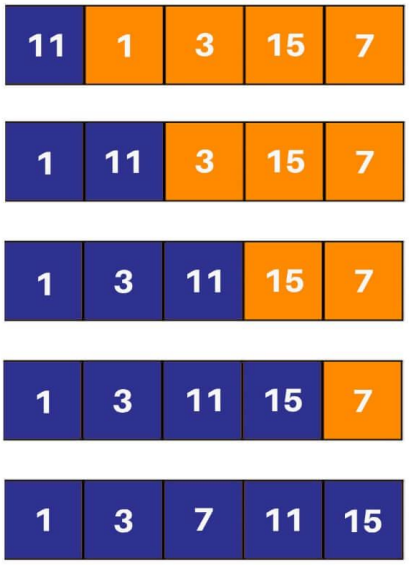
5. If the current element in the inner loop is smaller than the marked minimum element, change the value of the minimum element to the current element.
6. Swap the value of the minimum element with the element marked by the outer loop variable.
7. END

Selection Sort Java Code:

```
class Sort

{

    void selectionSort(int arr[])

    {

        int pos;

        int temp;

        for (int i = 0; i < arr.length; i++)

        {

            pos = i;

            for (int j = i+1; j < arr.length; j++)

           {

                if (arr[j] < arr[pos])                  //find the index of
the minimum element

                {

                    pos = j;

                }

            }
```

```
            temp = arr[pos];             //swap the current element with the
minimum element
            arr[pos] = arr[i];
            arr[i] = temp;
        }
    }

    void display(int arr[])                         //display the array
    {
        for (int i=0; i<arr.length; i++)
        {
            System.out.print(arr[i]+" ");
        }
    }

    public static void main(String args[])
    {
        Sort ob = new Sort();
        int arr[] = {64,25,12,22,11};
        ob.selectionSort(arr);
        ob.display(arr);
    }
}
```

Explanation of how it works:




## Merge Sort

Merge sort is one of the most flexible sorting algorithms in java known to mankind (yes, no kidding). It uses the divide and conquers strategy for sorting elements in an array. It is also a stable sort, meaning that it will not change the order of the original elements in an array concerning each other. The underlying strategy breaks up the array into multiple smaller segments till segments of only two elements (or one element) are obtained. Now, elements in these segments are sorted and the segments are merged to form larger segments. This process continues till the entire array is sorted.

This algorithm has two main parts:

- mergeSort() – This function calculates the middle index for the subarray and then partitions the subarray into two halves. The first half runs from index left to middle, while the second half runs from index middle+1 to right. After the partitioning is done, this function automatically calls the merge() function for sorting the subarray being handled by the mergeSort() call.
- merge() – This function does the actual heavy lifting for the sorting process. It requires the input of four parameters – the array, the starting index (left), the middle index (middle), and

the ending index (right). Once received, merge() will split the subarray into two subarrays – one left subarray and one right subarray. The left subarray runs from index left to middle, while the right subarray runs from index middle+1 to right. This function then merges the two subarrays to get the sorted subarray.

Merge Sort Java Code:

```
class Sort

{

    void merge(int arr[], int left, int middle, int right)

    {

        int low = middle - left + 1;                    //size of the left
subarray

        int high = right - middle;                      //size of the right
subarray



        int L[] = new int[low];                             //create the
left and right subarray

        int R[] = new int[high];



        int i = 0, j = 0;



        for (i = 0; i < low; i++)                                   //copy
elements into left subarray

        {

            L[i] = arr[left + i];

        }
```

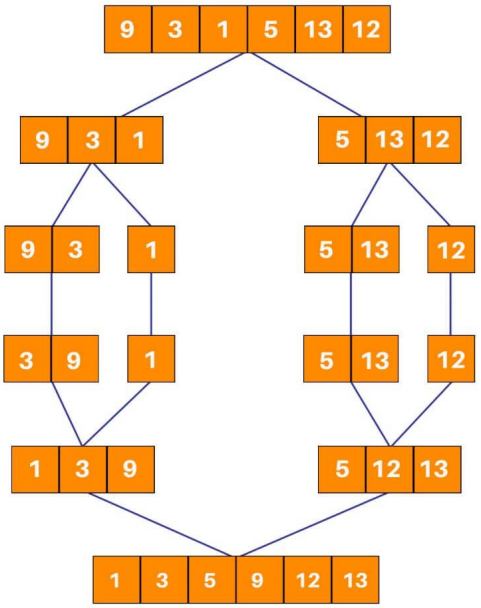
```
        for (j = 0; j < high; j++)                                  //copy
elements into right subarray
        {
            R[j] = arr[middle + 1 + j];
        }



        int k = left;                                               //get
starting index for sort
        i = 0;                                                  //reset loop
variables before performing merge
        j = 0;


        while (i < low && j < high)                         //merge the left and
right subarrays
        {
            if (L[i] <= R[j])
            {
                arr[k] = L[i];
                i++;
            }
            else
            {
                arr[k] = R[j];
```

```
                j++;

            }

            k++;

        }



        while (i < low)                                 //merge the remaining
elements from the left subarray

        {

            arr[k] = L[i];

            i++;

            k++;

        }



        while (j < high)                                //merge the remaining
elements from right subarray

        {

            arr[k] = R[j];

            j++;

            k++;

        }

    }
```

```
    void mergeSort(int arr[], int left, int right)       //helper function
that creates the sub cases for sorting
    {
        int middle;
        if (left < right) {                              //sort only if the
left index is lesser than the right index (meaning that sorting is done)
            middle = (left + right) / 2;


            mergeSort(arr, left, middle);                    //left subarray
            mergeSort(arr, middle + 1, right);               //right subarray


            merge(arr, left, middle, right);                //merge the two
subarrays
        }
    }


    void display(int arr[])                  //display the array
    {
        for (int i=0; i<arr.length; ++i)
        {
            System.out.print(arr[i]+" ");
        }
    }
```

```
    public static void main(String args[])

    {

        int arr[] = { 9, 3, 1, 5, 13, 12 };

        Sort ob = new Sort();

        ob.mergeSort(arr, 0, arr.length - 1);

        ob.display(arr);

    }

}
```

**Explanation of how it works:**




## Heap Sort

Heap sort is one of the most important sorting methods in java that one needs to learn to get into sorting. It combines the concepts of a tree as well as sorting, properly reinforcing the use of concepts

from both. A heap is a complete binary search tree where items are stored in a special order depending on the requirement. A min-heap contains the minimum element at the root, and every child of the root must be greater than the root itself. The children at the level after that must be greater than these children, and so on. Similarly, a max-heap contains the maximum element at the root. For the sorting process, the heap is stored as an array where for every parent node at the index i, the left child is at index 2 * i + 1, and the right child is at index 2 * i + 2.

A max heap is built with the elements of the unsorted array, and then the maximum element is extracted from the root of the array and then exchanged with the last element of the array. Once done, the max heap is rebuilt for getting the next maximum element. This process continues till there is only one node present in the heap.

This algorithm has two main parts:-

- heapSort() – This function helps construct the max heap initially for use. Once done, every root element is extracted and sent to the end of the array. Once done, the max heap is reconstructed from the root. The root is again extracted and sent to the end of the array, and hence the process continues.
- heapify() – This function is the building block of the heap sort algorithm. This function determines the maximum from the element being examined as the root and its two children. If the maximum is among the children of the root, the root and its child are swapped. This process is then repeated for the new root. When the maximum element in the array is found (such that its children are smaller than it) the function stops. For the node at index i, the left child is at index 2 * i + 1, and the right child is at index 2 * i + 1. (indexing in an array starts from 0, so the root is at 0).

### Heap Sort Java Code:

```
class Sort {

    public void heapSort(int arr[])

    {

        int temp;
```

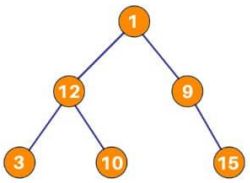
```
        for (int i = arr.length / 2 - 1; i >= 0; i--)                  //build
the heap
        {
            heapify(arr, arr.length, i);
        }

        for (int i = arr.length - 1; i > 0; i--)
//extract elements from the heap
        {
            temp = arr[0];
//move current root to end (since it is the largest)
            arr[0] = arr[i];
            arr[i] = temp;
            heapify(arr, i, 0);
//recall heapify to rebuild heap for the remaining elements
        }
    }

    void heapify(int arr[], int n, int i)
    {
        int MAX = i; // Initialize largest as root
        int left = 2 * i + 1; //index of the left child of ith node = 2*i + 1
        int right = 2 * i + 2; //index of the right child of ith node  = 2*i
+ 2
        int temp;
```

```
        if (left < n && arr[left] > arr[MAX])            //check if the left
child of the root is larger than the root
        {
            MAX = left;
        }

        if (right < n && arr[right] > arr[MAX])            //check if the
right child of the root is larger than the root
        {
            MAX = right;
        }

        if (MAX != i)
        {                                                  //repeat the
procedure for finding the largest element in the heap
            temp = arr[i];
            arr[i] = arr[MAX];
            arr[MAX] = temp;
            heapify(arr, n, MAX);
        }
    }

    void display(int arr[])                    //display the array
```

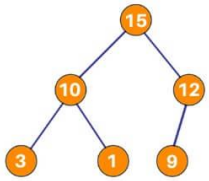
```
    {
        for (int i=0; i<arr.length; ++i)
        {
            System.out.print(arr[i]+" ");
        }
    }

    public static void main(String args[])
    {
        int arr[] = { 1, 12, 9 , 3, 10, 15 };

        Sort ob = new Sort();
        ob.heapSort(arr);
        ob.display(arr);
    }
}
```

### Explanation of how it works:

| 1 | 12 | 9 | 3 | 10 | 15 |
|---|---|---|---|---|---|




After construction of max heap:




| 15 | 10 | 12 | 3 | 1 | 9 |
|---|---|---|---|---|---|

| 9 | 10 | 12 | 3 | 1 | 15 |
|---|---|---|---|---|---|

Again, after construction of max heap:

| 12 | 10 | 9 | 3 | 1 | 15 |
|---|---|---|---|---|---|

| 1 | 10 | 9 | 3 | 12 | 15 |
|---|---|---|---|---|---|

Again, after construction of max heap:

| 10 | 9 | 1 | 3 | 12 | 15 |
|---|---|---|---|---|---|

| 3 | 9 | 1 | 10 | 12 | 15 |
|---|---|---|---|---|---|

Again, after construction of max heap:

| 9 | 1 | 3 | 10 | 12 | 15 |
|---|---|---|---|---|---|

| 3 | 1 | 9 | 10 | 12 | 15 |
|---|---|---|---|---|---|

Again, after construction of max heap:

| 3 | 1 | 9 | 10 | 12 | 15 |
|---|---|---|---|---|---|

| 1 | 3 | 9 | 10 | 12 | 15 |
|---|---|---|---|---|---|

## Doon Coding Questions

**#Write a Java Program to remove all white spaces from a string with using replace().**

**Answer:** This is a simple program where we have our string variable str1.

Another string variable str2 is initialized with the replaceAll option which is an inbuilt method to remove n number of whitespaces. Ultimately, we have printed str2 which has no whitespaces.

```
class RemoveWhiteSpaces

{

    public static void main(String[] args)

    {

        String str1 = "Saket Saurav     is a QualityAna     list";


        //1. Using replaceAll() Method


        String str2 = str1.replaceAll("\\s", "");


        System.out.println(str2);


           }

}

}
```

**Output:**

SaketSauravisaQualityAnalist

**Q Write a Java Program to remove all white spaces from a string without using replace().**

**Answer:** This is another approach to removing all the white spaces. Again, we have one string variable str1 with some value. Then, we have converted that string into a character array using toCharArray().

Then, we have one StringBuffer object sb which will be used to append the value stored at chars[i] index after we have included for loop and one if condition.

If the condition is set such that then the element at i index of the character array should not be equal to space or tab. Finally, we have printed our StringBuffer object sb.

```
class RemoveWhiteSpaces

{

    public static void main(String[] args)

    {

        String str1 = "Saket Saurav     is an Autom ation Engi ne         er";



        char[] chars = str1.toCharArray();



        StringBuffer sb = new StringBuffer();



        for (int i = 0; i &lt; chars.length; i++)

        {

            if( (chars[i] != ' ') &amp;&amp; (chars[i] != '\t') )

            {

                sb.append(chars[i]);

            }

        }

        System.out.println(sb);          //Output :
CoreJavajspservletsjdbcstrutshibernatespring

    }
```

```
}
```

**Output:**

SaketSauravisanAutomationEngineer

## Ajj ka gyan

What is the difference between a function and a method?

Ans). A method is a function that is written in a class. We do not have functions in java; instead we have methods. This means whenever a function is written in java, it should be written inside the class only. But if we take C++, we can write the functions inside as well as outside the class. So in C++, they are called member functions and not methods.

3) Which part of JVM will allocate the memory for a java program?

Ans). Class loader subsystem of JVM will allocate the necessary memory needed by the java program.

4). which algorithm is used by garbage collector to remove the unused variables or objects from memory?

Ans). Garbage collector uses many algorithms but the most commonly used algorithm is mark and sweep.

5). How can you call the garbage collector?

Ans). Garbage collector is automatically invoked when the program is being run. It can be also called by calling gc() method of Runtime class or System class in Java.

6) What is JIT Compiler?

Ans). JIT compiler is the part of JVM which increases the speed of execution of a Java program.

What is the difference between #include and import statement?

Ans). #include directive makes the compiler go to the C/C++ standard library and copy the code from the header files into the program. As a result, the program size increases, thus wasting memory and processor's time.

import statement makes the JVM go to the Java standard library, execute the code there , and substitute the result into the program. Here, no code is copied and hence no waste of memory or processor's time. so import is an efficient mechanism than #include.

What happens if String args[] is not written in main( ) method ?

Ans). When main( ) method is written without String args[] as:

Public static void main( )

The code will compile but JVM cannot run the code because it cannot recognize the main( ) as the method from where it should start execution of the Java program. Remember JVM always looks for main( ) method with string type array as parameter.

## Fun Facts To Know

is the difference between System.out.exit(0) and System.exit(1) ?

Ans). System.exit(0) terminates the program normally. Whereas System.exit(1) terminates the program because of some error encountered in the program.