Breath and Restart

Sirf khwabh Nhi

26/08

1] Java Class and Objects

the concept of classes and objects in Java with the help of examples.

Java is an object-oriented programming language. The core concept of the object-oriented approach is to break complex problems into smaller objects.

An object is any entity that has a state and behavior. For example, a `bicycle` is an object. It has

- States: idle, first gear, etc
- Behaviors: braking, accelerating, etc.

Before we learn about objects, let's first know about classes in Java.

Java Class

A class is a blueprint for the object. Before we create an object, we first need to define the class.

We can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

Since many houses can be made from the same description, we can create many objects from a class.

Create a class in Java

We can create a class in Java using the class keyword. For example,

```
class ClassName {

    // fields

    // methods
}
```

Here, `fields` (variables) and `methods` represent the state and behavior of the object respectively.

- fields are used to store data
- methods are used to perform some operations

For our `bicycle` object, we can create the class as

```
class Bicycle {


  // state or field
  private int gear = 5;


  // behavior or method
  public void braking() {
    System.out.println("Working of Braking");
  }
}
```

In the above example, we have created a class named `Bicycle`. It contains a field named `gear` and a method named `braking()`.

Here, `Bicycle` is a prototype. Now, we can create any number of bicycles using the prototype. And, all the bicycles will share the fields and methods of the prototype.

> Note: We have used keywords `private` and `public`. These are known as access modifiers.

Java Objects

An object is called an instance of a class. For example, suppose `Bicycle` is a class then `MountainBicycle`, `SportsBicycle`, `TouringBicycle`, etc can be considered as objects of the class.

Creating an Object in Java

Here is how we can create an object of a class.

```
className object = new className();



// for Bicycle class

Bicycle sportsBicycle = new Bicycle();



Bicycle touringBicycle = new Bicycle();
```

We have used the `new` keyword along with the constructor of the class to create an object. Constructors are similar to methods and have the same name as the class. For example, `Bicycle()` is the constructor of the `Bicycle` class.

Here, `sportsBicycle` and `touringBicycle` are the names of objects. We can use them to access fields and methods of the class.

As you can see, we have created two objects of the class. We can create multiple objects of a single class in Java.

> Note: Fields and methods of a class are also called members of the class.

Access Members of a Class

We can use the name of objects along with the `.` operator to access members of a class. For example,

```
class Bicycle {
    // field of class
    int gear = 5;


    // method of class
    void braking() {
        ...
    }
}


// create object
Bicycle sportsBicycle = new Bicycle();


// access field and method
sportsBicycle.gear;
sportsBicycle.braking();
```

In the above example, we have created a class named `Bicycle`. It includes a field named `gear` and a method named `braking()`. Notice the statement,

```
Bicycle sportsBicycle = new Bicycle();
```

Here, we have created an object of `Bicycle` named `sportsBicycle`. We then use the object to access the field and method of the class.

- sportsBicycle.gear - access the field `gear`
- sportsBicycle.braking() - access the method `braking()`

Now that we understand what is class and object. Let's see a fully working example.

Example: Java Class and Objects

```
class Lamp {

  // stores the value for light
  // true if light is on
  // false if light is off
  boolean isOn;

  // method to turn on the light
  void turnOn() {
    isOn = true;
    System.out.println("Light on? " + isOn);

  }

  // method to turnoff the light
  void turnOff() {
    isOn = false;
    System.out.println("Light on? " + isOn);
  }
}
```

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

    // create objects led and halogen
    Lamp led = new Lamp();
    Lamp halogen = new Lamp();

    // turn on the light by
    // calling method turnOn()
    led.turnOn();

    // turn off the light by
    // calling method turnOff()
    halogen.turnOff();
  }
}
```

Output:

```
Light on? true
Light on? false
```

In the above program, we have created a class named `Lamp`. It contains a variable: `isOn` and two methods: `turnOn()` and `turnOff()`.

Inside the `Main` class, we have created two objects: `led` and `halogen` of the `Lamp` class. We then used the objects to call the methods of the class.

- led.turnOn() - It sets the `isOn` variable to `true` and prints the output.
- halogen.turnOff() - It sets the `isOn` variable to `false` and prints the output.

The variable `isOn` defined inside the class is also called an instance variable. It is because when we create an object of the class, it is called an instance of the class. And, each instance will have its own copy of the variable.

That is, `led` and `halogen` objects will have their own copy of the `isOn` variable.

Example: Create objects inside the same class

Note that in the previous example, we have created objects inside another class and accessed the members from that class.

However, we can also create objects inside the same class.

```
class Lamp {



  // stores the value for light

  // true if light is on

  // false if light is off

  boolean isOn;



  // method to turn on the light

  void turnOn() {

    isOn = true;

    System.out.println("Light on? " + isOn);
    }

    public static void main(String[] args) {

        // create an object of Lamp
        Lamp led = new Lamp();

        // access method using object
        led.turnOn();
    }
}
```

Output

```
Light on? true
```

Here, we are creating the object inside the `main()` method of the same class.

Java Methods

we will learn about Java methods, how to define methods, and how to use methods in Java programs with the help of examples.

Java Methods

A method is a block of code that performs a specific task.

Suppose you need to create a program to create a circle and color it. You can create two methods to solve this problem:

- a method to draw the circle

- a method to color the circle

Dividing a complex problem into smaller chunks makes your program easy to understand and reusable.

In Java, there are two types of methods:

- User-defined Methods: We can create our own method based on our requirements.
- Standard Library Methods: These are built-in methods in Java that are available to use.

Let's first learn about user-defined methods.

Declaring a Java Method

The syntax to declare a method is:

```
returnType methodName() {

  // method body

}
```

Here,

- returnType - It specifies what type of value a method returns For example if a method has an `int` return type then it returns an integer value.

  If the method does not return a value, its return type is `void`.
- methodName - It is an identifier that is used to refer to the particular method in a program.
- method body - It includes the programming statements that are used to perform some tasks. The method body is enclosed inside the curly braces `{ }`.

For example,

```
int addNumbers() {
```

```
// code
}
```

In the above example, the name of the method is `adddNumbers()`. And, the return type is `int`. We will learn more about return types later in this tutorial.

This is the simple syntax of declaring a method. However, the complete syntax of declaring a method is

```
modifier static returnType nameOfMethod (parameter1, parameter2, ...) {
   // method body
}
```

Here,

- modifier - It defines access types whether the method is public, private, and so on.
- static - If we use the `static` keyword, it can be accessed without creating objects.

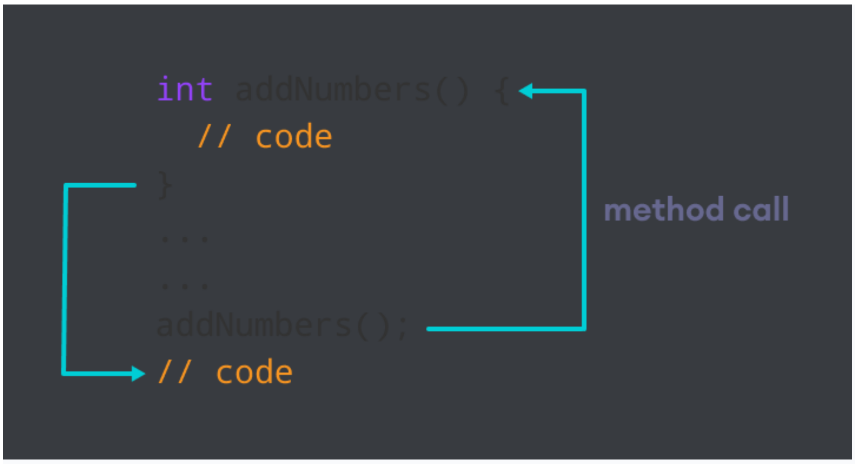
  For example, the `sqrt()` method of standard Math class is static. Hence, we can directly call `Math.sqrt()` without creating an instance of `Math` class.
- parameter1/parameter2 - These are values passed to a method. We can pass any number of arguments to a method.

Calling a Method in Java

In the above example, we have declared a method named `addNumbers()`. Now, to use the method, we need to call it.

Here's is how we can call the `addNumbers()` method.

```
// calls the method
addNumbers();
```



```
int addNumbers() {
  // code
}
...
...
addNumbers();
// code
```

Working of Java Method Call

Example 1: Java Methods

```
class Main {


  // create a method

  public int addNumbers(int a, int b) {

    int sum = a + b;

    // return value

    return sum;

  }


  public static void main(String[] args) {


    int num1 = 25;

    int num2 = 15;


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

    // calling method

    int result = obj.addNumbers(num1, num2);

    System.out.println("Sum is: " + result);

  }

}
```

Output

```
Sum is: 40
```

In the above example, we have created a method named `addNumbers()`. The method takes two parameters `a` and `b`. Notice the line,

```
int result = obj.addNumbers(num1, num2);
```

Here, we have called the method by passing two arguments `num1` and `num2`. Since the method is returning some value, we have stored the value in the `result` variable.

> Note: The method is not static. Hence, we are calling the method using the object of the class.

Java Method Return Type

A Java method may or may not return a value to the function call. We use the return statement to return any value. For example,

```
int addNumbers() {

...

return sum;

}
```

Here, we are returning the variable `sum`. Since the return type of the function is `int`. The sum variable should be of `int` type. Otherwise, it will generate an error.

Example 2: Method Return Type

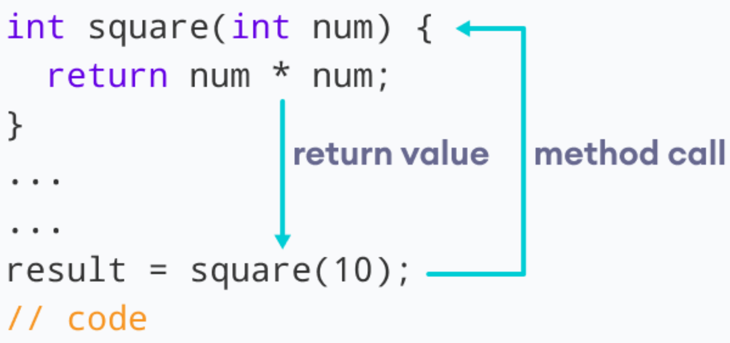
```
class Main {

// create a method
  public static int square(int num) {

    // return statement
    return num * num;
  }

  public static void main(String[] args) {
    int result;

    // call the method
    // store returned value to result
    result = square(10);

    System.out.println("Squared value of 10 is: " + result);
  }
}
```

Output:

```
Squared value of 10 is: 100
```

In the above program, we have created a method named `square()`. The method takes a number as its parameter and returns the square of the number.

Here, we have mentioned the return type of the method as `int`. Hence, the method should always return an integer value.



```
int square(int num) {
  return num * num;
}
...
...
result = square(10);
// code
```

Representation of the Java method returning a value

Note: If the method does not return any value, we use the void keyword as the return type of the method. For example,

```
public void square(int a) {

  int square = a * a;

  System.out.println("Square is: " + a);

}
```

Method Parameters in Java

A method parameter is a value accepted by the method. As mentioned earlier, a method can also have any number of parameters. For example,

```
// method with two parameters

int addNumbers(int a, int b) {

  // code
}


// method with no parameter

int addNumbers(){

  // code

}
```

If a method is created with parameters, we need to pass the corresponding values while calling the method. For example,

```
// calling the method with two parameters

addNumbers(25, 15);


// calling the method with no parameters

addNumbers()
```

Example 3: Method Parameters

```
class Main {


  // method with no parameter

  public void display1() {

    System.out.println("Method without parameter");

  }


  // method with single parameter

  public void display2(int a) {
    System.out.println("Method with a single parameter: " + a);
  }

  public static void main(String[] args) {

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

    // calling method with no parameter
    obj.display1();

    // calling method with the single parameter
    obj.display2(24);
  }
}
```

Output

```
Method without parameter
Method with a single parameter: 24
```

Here, the parameter of the method is `int`. Hence, if we pass any other data type instead of `int`, the compiler will throw an error. It is because Java is a strongly typed language.

Note: The argument `24` passed to the `display2()` method during the method call is called the actual argument.

The parameter `num` accepted by the method definition is known as a formal argument. We need to specify the type of formal arguments. And, the type of actual arguments and formal arguments should always match.

Standard Library Methods

The standard library methods are built-in methods in Java that are readily available for use. These standard libraries come along with the Java Class Library (JCL) in a Java archive (*.jar) file with JVM and JRE.

For example,

- `print()` is a method of `java.io.PrintSteam`. The `print("...")` method prints the string inside quotation marks.
- `sqrt()` is a method of `Math` class. It returns the square root of a number.

Here's a working example:

Example 4: Java Standard Library Method

```
public class Main {

  public static void main(String[] args) {



    // using the sqrt() method

    System.out.print("Square root of 4 is: " + Math.sqrt(4));

  }

}
```

Output:

```
Square root of 4 is: 2.0
```

What are the advantages of using methods?

1. The main advantage is code reusability. We can write a method once, and use it multiple times. We do not have to rewrite the entire code each time. Think of it as, "write once, reuse multiple times".

Example 5: Java Method for Code Reusability

```
public class Main {

  // method defined
  private static int getSquare(int x){
    return x * x;
  }

  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) {

      // method call
      int result = getSquare(i);
      System.out.println("Square of " + i + " is: " + result);
    }
  }
}
```

Output:

```
Square of 1 is: 1
Square of 2 is: 4
Square of 3 is: 9

Square of 4 is: 16

Square of 5 is: 25
```

In the above program, we have created the method named `getSquare()` to calculate the square of a number. Here, the method is used to calculate the square of numbers less than 6.

Hence, the same method is used again and again.

2. Methods make code more readable and easier to debug. Here, the `getSquare()` method keeps the code to compute the square in a block. Hence, makes it more readable.

Java Method Overloading

method overloading and how you can achieve it in Java with the help of examples.

In Java, two or more methods may have the same name if they differ in parameters (different number of parameters, different types of parameters, or both). These methods are called overloaded methods and this feature is called method overloading. For example:

```
void func() { ... }

void func(int a) { ... }

float func(double a) { ... }

float func(int a, float b) { ... }
```

Here, the `func()` method is overloaded. These methods have the same name but accept different arguments.

> Note: The return types of the above methods are not the same. It is because method overloading is not associated with return types. Overloaded methods may have the same or different return types, but they must differ in parameters.

Why method overloading?

Suppose, you have to perform the addition of given numbers but there can be any number of arguments (let's say either 2 or 3 arguments for simplicity).

In order to accomplish the task, you can create two methods `sum2num(int, int)` and `sum3num(int, int, int)` for two and three parameters respectively. However, other programmers, as well as you in the future may get confused as the behavior of both methods are the same but they differ by name.

The better way to accomplish this task is by overloading methods. And, depending upon the argument passed, one of the overloaded methods is called. This helps to increase the readability of the program.

How to perform method overloading in Java?

Here are different ways to perform method overloading:

1. Overloading by changing the number of parameters

```
class MethodOverloading {

    private static void display(int a){

        System.out.println("Arguments: " + a);

    }



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

        System.out.println("Arguments: " + a + " and " + b);

    }



    public static void main(String[] args) {

        display(1);

        display(1, 4);

    }
}
```

Output:

```
Arguments: 1

Arguments: 1 and 4
```

2. Method Overloading by changing the data type of parameters

```
class MethodOverloading {



    // this method accepts int

    private static void display(int a){

        System.out.println("Got Integer data.");

    }



    // this method  accepts String object

    private static void display(String a){

        System.out.println("Got String object.");

    }



    public static void main(String[] args) {

        display(1);

        display("Hello");

    }

}
```

Output:

```
Got Integer data.

Got String object.
```

Here, both overloaded methods accept one argument. However, one accepts the argument of type `int` whereas other accepts `String` object.

Let’s look at a real-world example:

```
class HelperService {


    private String formatNumber(int value) {

        return String.format("%d", value);

    }


    private String formatNumber(double value) {

        return String.format("%.3f", value);

    }


    private String formatNumber(String value) {

        return String.format("%.2f", Double.parseDouble(value));

    }


    public static void main(String[] args) {

        HelperService hs = new HelperService();

        System.out.println(hs.formatNumber(500));
        System.out.println(hs.formatNumber(89.9934));

        System.out.println(hs.formatNumber("550"));

    }

}
```

When you run the program, the output will be:

```
500

89.993

550.00
```

> Note: In Java, you can also overload constructors in a similar way like methods.

Important Points

- Two or more methods can have the same name inside the same class if they accept different arguments. This feature is known as method overloading.
- Method overloading is achieved by either:
  - changing the number of arguments.
  - or changing the data type of arguments.
- It is not method overloading if we only change the return type of methods. There must be differences in the number of parameters.

Java Constructors

we will learn about Java constructors, their types, and how to use them with the help of examples.

What is a Constructor?

A constructor in Java is similar to a method that is invoked when an object of the class is created.

Unlike Java methods, a constructor has the same name as that of the class and does not have any return type. For example,

```
class Test {

  Test() {

    // constructor body

  }

}
```

Here, `Test()` is a constructor. It has the same name as that of the class and doesn't have a return type.

Example 1: Java Constructor

```
class Main {

  private String name;



  // constructor

  Main() {

    System.out.println("Constructor Called:");

    name = "Programiz";

  }
  public static void main(String[] args) {

    // constructor is invoked while
    // creating an object of the Main class
    Main obj = new Main();
    System.out.println("The name is " + obj.name);
  }
}
```

Output:

```
Constructor Called:
The name is Programiz
```

In the above example, we have created a constructor named `Main()`. Inside the constructor, we are initializing the value of the `name` variable.

Notice the statement of creating an object of the `Main` class.

```
Main obj = new Main();
```

Here, when the object is created, the `Main()` constructor is called. And, the value of the `name` variable is initialized.

Hence, the program prints the value of the `name` variables as `Programiz`.

Types of Constructor

In Java, constructors can be divided into 3 types:

1. No-Arg Constructor
2. Parameterized Constructor
3. Default Constructor

1. Java No-Arg Constructors

Similar to methods, a Java constructor may or may not have any parameters (arguments).

If a constructor does not accept any parameters, it is known as a no-argument constructor. For example,

```
private Constructor() {

    // body of the constructor

}
```

Example 2: Java private no-arg constructor

```
class Main {


  int i;


  // constructor with no parameter

  private Main() {

    i = 5;

    System.out.println("Constructor is called");

  }


  public static void main(String[] args) {


    // calling the constructor without any parameter

    Main obj = new Main();
    System.out.println("Value of i: " + obj.i);
  }
}
```

Output:

```
Constructor is called
Value of i: 5
```

In the above example, we have created a constructor `Main()`. Here, the constructor does not accept any parameters. Hence, it is known as a no-arg constructor.

Notice that we have declared the constructor as private.

Once a constructor is declared `private`, it cannot be accessed from outside the class. So, creating objects from outside the class is prohibited using the private constructor.

Here, we are creating the object inside the same class. Hence, the program is able to access the constructor.

However, if we want to create objects outside the class, then we need to declare the constructor as `public`.

Example 3: Java public no-arg constructors

```
class Company {
  String name;


  // public constructor
  public Company() {
    name = "Programiz";
  }
}
```

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

    // object is created in another class
    Company obj = new Company();
    System.out.println("Company name = " + obj.name);
  }
}
```

Output:

```
Company name = Programiz
```

2. Java Parameterized Constructor

A Java constructor can also accept one or more parameters. Such constructors are known as parameterized constructors (constructor with parameters).

Example 4: Parameterized constructor

```
class Main {

  String languages;

  // constructor accepting single value
  Main(String lang) {
    languages = lang;
    System.out.println(languages + " Programming Language");

  }



  public static void main(String[] args) {



    // call constructor by passing a single value

    Main obj1 = new Main("Java");

    Main obj2 = new Main("Python");

    Main obj3 = new Main("C");

  }

}
```

Output:

```
Java Programming Language

Python Programming Language

C Programming Language
```

In the above example, we have created a constructor named `Main()`. Here, the constructor takes a single parameter. Notice the expression,

```
Main obj1 = new Main("Java");
```

Here, we are passing the single value to the constructor. Based on the argument passed, the language variable is initialized inside the constructor.

---

3. Java Default Constructor

If we do not create any constructor, the Java compiler automatically create a no-arg constructor during the execution of the program. This constructor is called default constructor.

Example 5: Default Constructor

```
class Main {


  int a;

  boolean b;


  public static void main(String[] args) {


    // A default constructor is called

    Main obj = new Main();


    System.out.println("Default Value:");

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

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

  }

}
```

Output:

```
a = 0

b = false
```

Here, we haven't created any constructors. Hence, the Java compiler automatically creates the default constructor.

The default constructor initializes any uninitialized instance variables with default values.

Type	Default Value
`boolean`	false
`byte`	0
`short`	0
`int`	0
`long`	0L
`char`	\u0000
`float`	0.0f
`double`	0.0d
`object`	Reference null

In the above program, the variables `a` and `b` are initialized with default value 0 and `false` respectively.

The above program is equivalent to:

```
class Main {
  int a;
  boolean b;

  // a private constructor
  private Main() {
    a = 0;
    b = false;
  }

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

    System.out.println("Default Value:");
    System.out.println("a = " + obj.a);
    System.out.println("b = " + obj.b);
  }
}
```

The output of the program is the same as Example 5.

Important Notes on Java Constructors

- Constructors are invoked implicitly when you instantiate objects.
- The two rules for creating a constructor are:
  The name of the constructor should be the same as the class.
  A Java constructor must not have a return type.
- If a class doesn't have a constructor, the Java compiler automatically creates a default constructor during run-time. The default constructor initializes instance variables with default values. For example, the `int` variable will be initialized to `0`
- Constructor types:
  No-Arg Constructor - a constructor that does not accept any arguments
  Parameterized constructor - a constructor that accepts arguments
  Default Constructor - a constructor that is automatically created by the Java compiler if it is not explicitly defined.
- A constructor cannot be `abstract` or `static` or `final`.
- A constructor can be overloaded but can not be overridden.

Constructors Overloading in Java

Similar to Java method overloading, we can also create two or more constructors with different parameters. This is called constructors overloading.

Example 6: Java Constructor Overloading

```
class Main {


  String language;


  // constructor with no parameter

  Main() {

    this.language = "Java";

  }
  // constructor with a single parameter
  Main(String language) {
    this.language = language;
  }

  public void getName() {
    System.out.println("Programming Langauage: " + this.language);
  }

  public static void main(String[] args) {

    // call constructor with no parameter
    Main obj1 = new Main();

    // call constructor with a single parameter
    Main obj2 = new Main("Python");

    obj1.getName();
    obj2.getName();
  }
}
```

Output:

```
Programming Language: Java
```

```
Programming Language: Python
```

In the above example, we have two constructors: `Main()` and `Main(String language)`. Here, both the constructor initialize the value of the variable language with different values.

Based on the parameter passed during object creation, different constructors are called and different values are assigned.

It is also possible to call one constructor from another constructor.

> Note: We have used `this` keyword to specify the variable of the class.

IMP TECHKNOWLEDGE

1)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.

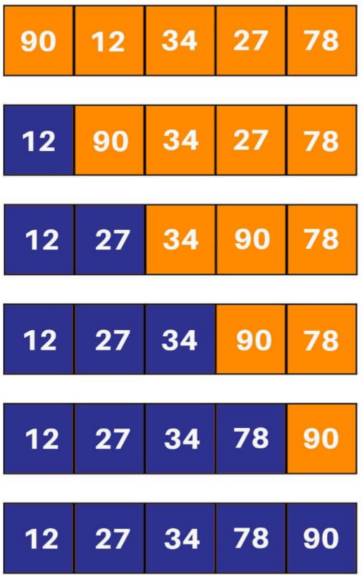
Algorithm:

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:

```java
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:



90 12 34 27 78

12 90 34 27 78

12 27 34 90 78

12 27 34 90 78

12 27 34 78 90

12 27 34 78 90

2) 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.

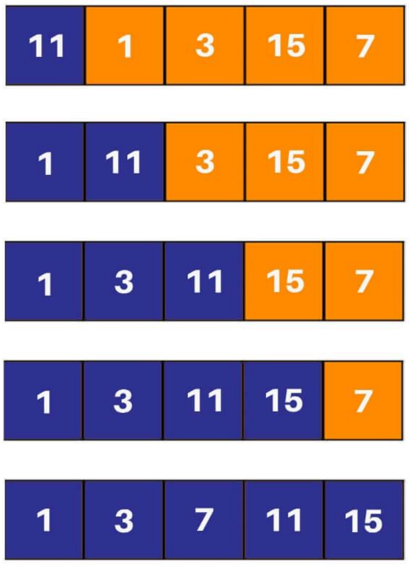
Algorithm:

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:



11 1 3 15 7

1 11 3 15 7

1 3 11 15 7

1 3 11 15 7

1 3 7 11 15

REVISE kaar heee

1) 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.

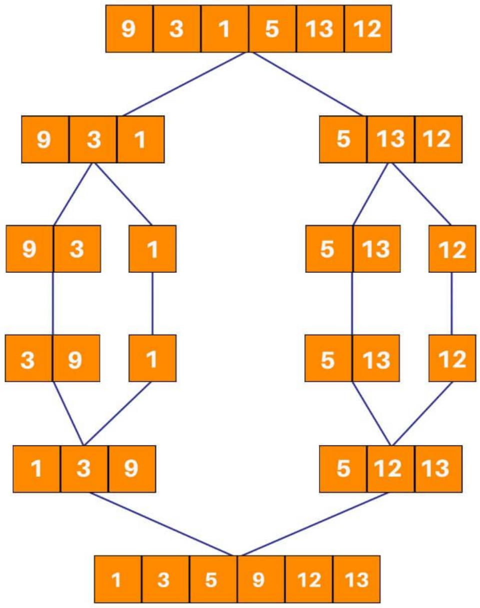
Merge Sort Java Code:

```
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:



2) 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:-

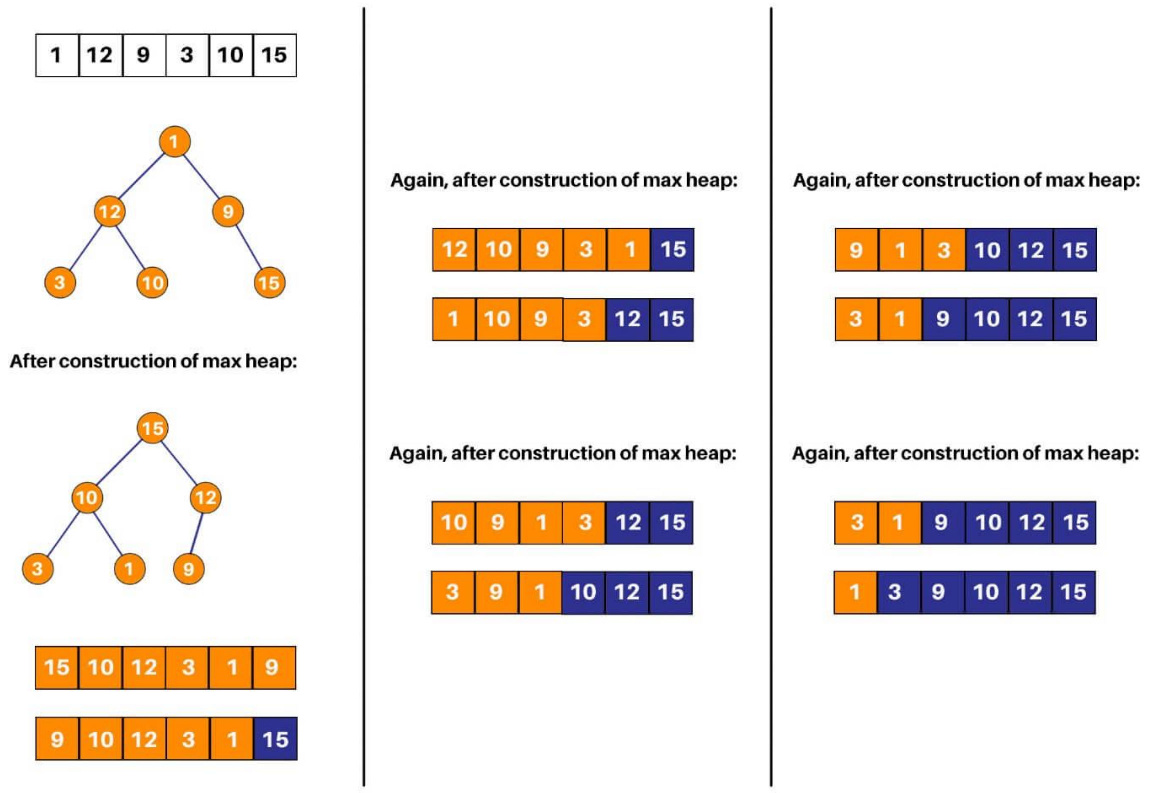
- heapSort() – This function helps construct the max heap initially for use. Once done, every root element is extracted and sent to the end of the array. Once done, the max heap is reconstructed from the root. The root is again extracted and sent to the end of the array, and hence the process continues.
- 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:

```java
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
1
12
9
3
10
15
After construction of max heap:
15
10
12
3
1
9
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 swap two numbers using the third variable.

Answer: In this example, we have made use of the Scanner class to declare an object with a predefined standard input object. This program will accept the values of x and y through the command line (when executed).

We have used nextInt() which will input the value of an integer variable ‘x’ and ‘y’ from the user. A temp variable is also declared.

Now, the logic of the program goes like this – we are assigning temp or third variable with the value of x, and then assigning x with the value of y and again assigning y with the value of temp. So, after the first complete iteration, the temp will have a value of x, x will have a value of y and y will have a value of temp (which is x).

```java
import java.util.Scanner;

public class SwapTwoNumbers {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int x, y, temp;
        System.out.println("Enter x and y");
        Scanner in = new Scanner(System.in);
        x = in.nextInt();
        y = in.nextInt();
        System.out.println("Before Swapping" + x + y);
        temp = x;
        x = y;
        y = temp;
        System.out.println("After Swapping" + x + y);

    }

}
```

Output:

```
Enter x and y

45

98

Before Swapping4598

After Swapping9845
```

# Write a Java Program to swap two numbers without using the third variable.

Answer: Rest all things will be the same as the above program. Only the logic will change. Here, we are assigning x with the value x + y which means x will have a sum of both x and y.

Then, we are assigning y with the value x – y which means we are subtracting the value of y from the sum of (x + y). Till here, x still has the sum of both x and y. But y has the value of x.

Finally, in the third step, we are assigning x with the value x – y which means we are subtracting y (which has the value of x) from the total (x + y). This will assign x with the value of y and vice versa.

```
import java.util.Scanner;

class SwapTwoNumberWithoutThirdVariable
{
   public static void main(String args[])
   {
      int x, y;
      System.out.println("Enter x and y");
      Scanner in = new Scanner(System.in);

      x = in.nextInt();
      y = in.nextInt();

      System.out.println("Before Swapping\nx = "+x+"\ny = "+y);

      x = x + y;
      y = x - y;
      x = x - y;

      System.out.println("After Swapping without third variable\nx =
"+x+"\ny = "+y);
   }
}
```

Output:

Enter x and y

45

98

Before Swapping

x = 45

y = 98

After Swapping without a third variable

x = 98

y = 45

# Write a Java Program to count the number of words in a string using HashMap.

Answer: This is a collection class program where we have used HashMap for storing the string.

First of all, we have declared our string variable called str. Then we have used split() function delimited by single space so that we can split multiple words in a string.

Thereafter, we have declared HashMap and iterated using for loop. Inside for loop, we have an if-else statement in which wherever at a particular position, the map contains a key, we set the counter at that position and add the object to the map.

Each time, the counter is incremented by 1. Else, the counter is set to 1.

Finally, we are printing the HashMap.

Note: The same program can be used to count the number of characters in a string. All you need to do is to remove one space (remove space delimited in split method) in String[] split = str.split("");

```
import java.util.HashMap;

public class FinalCountWords {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        String str = "This this is is done by Saket Saket";
        String[] split = str.split(" ");

                HashMap<String,Integer> map = new
HashMap<String,Integer>();
        for (int i=0; i<split.length; i++) {
            if (map.containsKey(split[i])) {
                int count = map.get(split[i]);
                map.put(split[i], count+1);
            }
            else {
                map.put(split[i], 1);
            }
        }
        System.out.println(map);
    }

}
```

Output:

```
{Saket=2, by=1, this=1, This=1, is=2, done=1}
```

Ajj ka gyan

Why do constructors not return values?

What actually happens with the constructor is that the runtime uses type data generated by the compiler to determine how much space is needed to store an object instance in memory, be it on the stack or on the heap.
This space includes all members variables and the vtbl. After this space is allocated, the constructor is called as an internal part of the instantiation and initialization process to initialize the contents of the fields.
Then, when the constructor exits, the runtime returns the newly-created instance. So the reason the constructor doesn't return a value is because it's not called directly by your code, it's called by the memory allocation and object initialization code in the runtime.
Its return value (if it actually has one when compiled down to machine code) is opaque to the user - therefore, you can't specify it.

Well, in a way it returns the instance that has just been constructed.
You even call it like this, for example is Java
Object o = new Something();
which looks just like calling a "regular" method with a return value
Object o = someMethod();

Fun Facts To Know

#KAAHANI

Explain the difference between the following two statements:

1. String s=”Hello”

2. String s = new String (“Hello”);

Ans). In the first statement, assignment operator is used to assign the string literal to the String variables. In this case, JVM first of all checks whether the same object is already available in the string constant pool. If it is available, then it creates another reference to it. If the same object is not available, then it creates another object with the content “Hello “and stores it into the string constant pool.
In the second statement, new operator is used to create the string object; in this case, JVM always creates a new object without looking in the string constant pool.