Unit-2 Programming in Java
- Classes, Objects and Methods.
- Polymorphism: Method Overloading.
- Constructor: Concept of Constructor, Types of Constructor, Constructor Overloading.
- Garbage Collection, Finalize () Method.
- The **this** keyword.
- **static** and **final** keyword.
- Access Control: Public, Private, Protected

Java Classes/Objects

- Java is an object-oriented programming language.
- Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

Object in Java

- An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can. The example of an intangible object is the banking system.

An object has three characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used **internally by the JVM to identify each object uniquely.**

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

- An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

- An object is a real-world entity.
- An object is a runtime entity.
- The object is an entity which has state and behavior.
- The object is an instance of a class.

Java Classes

- A class in Java is a set of objects which shares common characteristics/ behavior and common properties/ attributes.
- It is a user-defined blueprint or prototype from which objects are created. For example, Student is a class while a particular student named Rajiv is an object.

Properties of Java Classes

1. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
2. Class does not occupy memory.
3. Class is a group of variables of different data types and a group of methods.
4. A Class in Java can contain:
   - Data member
   - Method
   - Constructor
   - Nested Class
   - Interface

Class Declaration in Java

```
access_modifier class <class_name>
{
   data member;
   method;
   constructor;
   nested class;
   interface;
}
```

Example

```
public class Main
 {
  int x = 5;

  public static void main(String[] args)
{
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

Multiple Objects

- You can create multiple objects of one class

```
public class Main
{
  int x = 5;
            public static void main(String[] args)
           {
                  Main myObj1 = new Main();  // Object 1
                  Main myObj2 = new Main();  // Object 2

                  System.out.println(myObj1.x);
                  System.out.println(myObj2.x);
           }
}
```

Using Multiple Classes

- You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the attributes and methods, while the other class holds the main() method (code to be executed)).
- Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory/folder:
    - Main.java
    - Second.java

Main.java

```
public class Main
{
  int x = 5;
}
```

Second.java

```
class Second
{
        public static void main(String[] args)
         {
                Main myObj = new Main();
                System.out.println(myObj.x);
         }
}
```

When both files have been compiled:

```
C:\Users\Your Name>javac Main.java
C:\Users\Your Name>javac Second.java
```

Run the Second.java file:

```
C:\Users\Your Name>java Second
```

And the output will be:

```
5
```

3 Ways to initialize object

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

1) Object and Class Example: Initialization through reference

- Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

TestStudent2.java

```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Sonoo";
  System.out.println(s1.id+" "+s1.name);//printing members with a white space
 }
}
```

Output:

```
101 Sonoo
```

We can also create multiple objects and store information in it through reference variable.

File: TestStudent3.java

```
class Student{
 int id;
 String name;
}
class TestStudent3{
 public static void main(String args[]){
  //Creating objects
  Student s1=new Student();
  Student s2=new Student();
  //Initializing objects
  s1.id=101;
  s1.name="Sonoo";
  s2.id=102;
  s2.name="Amit";
  //Printing data
  System.out.println(s1.id+" "+s1.name);
  System.out.println(s2.id+" "+s2.name);
 }
}
```

Output:

```
101 Sonoo
102 Amit
```

2) Object and Class Example: Initialization through method

- In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

File: TestStudent4.java

```
class Student
{
        int rollno;
        String name;
        void insertRecord(int r, String n)
        {
                 rollno=r;
                name=n;
        }
         void displayInformation()
        {
                System.out.println(rollno+" "+name);
        }
}
class TestStudent4
{
  public static void main(String args[])
  {
           Student s1=new Student();
          Student s2=new Student();
          s1.insertRecord(111,"Karan");
          s2.insertRecord(222,"Aryan");
          s1.displayInformation();
         s2.displayInformation();
 }
}
```

Output:

```
111 Karan
222 Aryan
```

3) Object and Class Example: Initialization through a constructor

- We will learn about constructors in Java later.

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:

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

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
}
```

Syntax

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

Here,

- **modifier** - It defines access types whether the method is public, private, and so on. To learn more, visit Java Access Specifier.
- **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();
```

Example 1:

```
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
```

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;
}
```

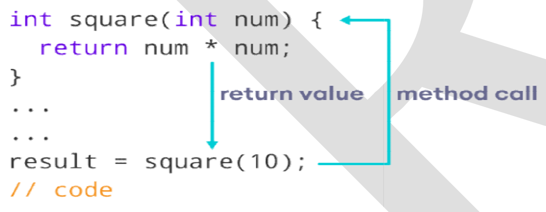
Here, we are returning the variable `sum`. Since the return type of the function is `int`. The sum variable should be of `int` type. Otherwise, it will generate an error.

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
```



```
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: " + square);
}
```

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
```

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.

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
```

Java Polymorphism

- Polymorphism is an important concept of object-oriented programming. It simply means more than one form.

Method Overloading in Java

- In Java, Method Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters, or a mixture of both.
- Method overloading in Java is also known as **Compile-time Polymorphism, Static Polymorphism, or Early binding**. In Method overloading compared to the parent argument, the child argument will get the highest priority.

```
// Java program to demonstrate working of method
// overloading in Java

public class Sum {
        // Overloaded sum(). This sum takes two int parameters
        public int sum(int x, int y) { return (x + y); }

        // Overloaded sum(). This sum takes three int parameters
        public int sum(int x, int y, int z)
        {
                return (x + y + z);
        }

        // Overloaded sum(). This sum takes two double
        // parameters
        public double sum(double x, double y)
        {
                return (x + y);
        }

        // Driver code
        public static void main(String args[])
        {
                Sum s = new Sum();
                System.out.println(s.sum(10, 20));
                System.out.println(s.sum(10, 20, 30));
                System.out.println(s.sum(10.5, 20.5));
        }
}
```

Constructor

- In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.
- It is a special type of method which is used to initialize the object.
- Every time an object is created using the new() keyword, at least one constructor is called.
- It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.
- There are two types of constructors in Java: no-arg constructor, and parameterized constructor.
- **Note:** It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor

There are two rules defined for the constructor.

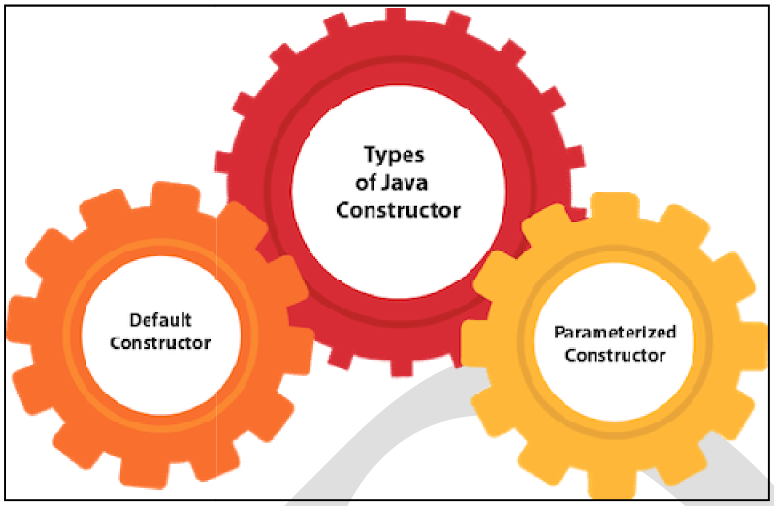
1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Note: We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor



1 Java Default Constructor)

- A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

```
<class_name>(){}
```

Example of default constructor

- In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
//Java Program to create and call a default constructor
class Bike1
{
        //creating a default constructor
        Bike1(){ System.out.println("Bike is created"); }
//main method
        public static void main(String args[])
        {
        //calling a default constructor
        Bike1 b=new Bike1();
        }
}
```

Output:

```
Bike is created
```

Example

```
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);
  }
}
```

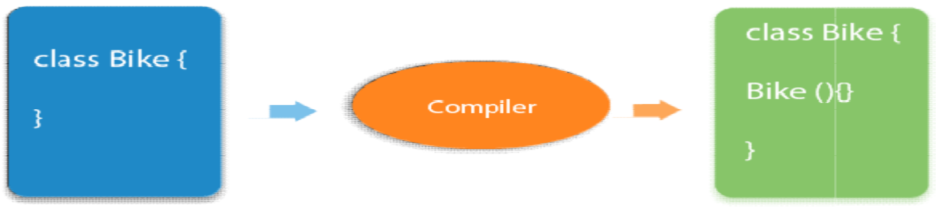
Output:

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

Example

```
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);
  }
}
```

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.



Example :

```
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:

```
Default Value:
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.

2)Java Parameterized Constructor

- A constructor which has a specific number of parameters is called a parameterized constructor.

Why use the parameterized constructor?

- The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example

```
//Java Program to demonstrate the use of the parameterized constructor.
class Student
{
   int id;
   String name;

      Student(int i,String n)
      {
                 id = i;
                 name = n;
   }
      void display()
 {
         System.out.println(id+" "+name);
 }

   public static void main(String args[])
   {
         Student s1 = new Student4(111,"Karan");
         Student s2 = new Student4(222,"Aryan");

           s1.display();
           s2.display();
  }
}
```

Output:

```
111 Karan
222 Aryan
```

Example

```
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
```

Example

Here is a simple example that uses a constructor −

```
// A simple constructor.
class MyClass {
  int x;

  // Following is the constructor
  MyClass(int i ) {
    x = i;
  }
}
```

You would call constructor to initialize objects as follows −

```
public class ConsDemo
{
  public static void main(String args[])
  {
    MyClass t1 = new MyClass( 10 );
    MyClass t2 = new MyClass( 20 );
    System.out.println(t1.x + " " + t2.x);
  }
}
```

Output

```
10 20
```

Constructor Overloading in Java

- In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.
- Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example

```
//Java program to overload constructors
class Student5{
   int id;
   String name;
   int age;

   Student5(int i,String n)
   {
   id = i;
   name = n;
   }

  Student5(int i,String n,int a)
  {
   id = i;
   name = n;
   age=a;
   }
   void display()
  {
          System.out.println(id+" "+name+" "+age);
   }

   public static void main(String args[])
 {
   Student5 s1 = new Student5(111,"Karan");
   Student5 s2 = new Student5(222,"Aryan",25);
   s1.display();
   s2.display();
  }
}
```

Output:

```
111 Karan 0
222 Aryan 25
```

Example

```
// Constructor Overloading
class Box
{
        double width, height, depth;
        Box(double w, double h, double d)
        {
                width = w;
                height = h;
                depth = d;
        }
        Box()
        {
            width = height = depth = 0;
        }
        Box(double len)
        {
             width = height = depth = len;
        }
        double volume()
        {
              return width * height * depth;
       }
}
public class Test
 {
        public static void main(String args[])
        {
                Box mybox1 = new Box(10, 20, 15);
                Box mybox2 = new Box();
                Box mycube = new Box(7);
                double vol;

               vol = mybox1.volume();
                System.out.println("Volume of mybox1 is " + vol);

                vol = mybox2.volume();
                System.out.println("Volume of mybox2 is " + vol);

                vol = mycube.volume();
                System.out.println("Volume of mycube is " + vol);
        } }
```

Example

```
class Rectangle
{
        public int width;
        public int height;

        // Constructor that takes width and height parameters
        public Rectangle(int width, int height)
       {
           this.width = width;
           this.height = height;
        }
        public Rectangle(int width)
       {
           this(width, width);
        }
}
class PrepBytes
{
    public static void main(String args[])
     {
           Rectangle rec1 = new Rectangle(5);
           System.out.println("Height of Rectangle is " + rec1.height);
           System.out.println("Width of Rectangle is " + rec1.width);
      }
}
```

Output:

```
Height of Rectangle is 5
Width of Rectangle is 5
```

Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.

Garbage Collection

- Garbage collection in Java is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short.

- When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

How Does Garbage Collection in Java works?

- Java garbage collection is an automatic process. Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects.
- An in-use object, or a referenced object, means that some part of your program still maintains a pointer to that object.
- An unused or unreferenced object is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed. The programmer does not need to mark objects to be deleted explicitly. The garbage collection implementation lives in the JVM.

Advantages of Garbage Collection in Java

- It makes java memory-efficient because the garbage collector removes the unreferenced objects from heap memory.

- It is automatically done by the garbage collector(a part of JVM), so we don’t need extra effort.

Finalize() method

- The Java **finalize() method** of Object class is a method that the Garbage Collector always calls just before the deletion/destroying the object which is eligible for Garbage Collection to perform clean-up activity.

- Clean-up activity means closing the resources associated with that object like Database Connection, Network Connection, or we can say resource de-allocation. Remember, it is not a reserved keyword. Once the finalize() method completes immediately, Garbage Collector destroys that object.

- Finalization in Java.

Note: The Garbage collector calls the finalize() method only once on any object.

Syntax:

```
protected void finalize throws Throwable{}
```

- Since the Object class contains the finalize method hence finalize method is available for every java class since Object is the superclass of all java classes. Since it is available for every java class, Garbage Collector can call the finalize() method on any java object.

Why finalize() method is used?

- finalize() method releases system resources before the garbage collector runs for a specific object. JVM allows finalize() to be invoked only once per object.

How to override finalize() method?

- The finalize method, which is present in the Object class, has an **empty implementation**. In our class, clean-up activities are there. Then we have to **override this method** to define our clean-up activities.

- In order to Override this method, we have to define and call finalize within our code explicitly.

```
import java.lang.*;
public class demo
{
        protected void finalize() throws Throwable
        {
                Try
                 {
                        System.out.println("inside demo's finalize()");
                   }
                catch (Throwable e)
                  {

                        throw e;
                }
                finally
                {

                        System.out.println("Calling finalize method"
                                                        + " of the Object class");

                        // Calling finalize() of Object class
                        super.finalize();
                }
        }

        // Driver code
        public static void main(String[] args) throws Throwable
        {

                // Creating demo's object
                demo d = new demo();

                // Calling finalize of demo
                d.finalize();
        }
}
```

This keyword

Definition and Usage

- In Java, this is a **reference variable** that refers to the current object.

- The most common use of the `this` keyword is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor parameter).

Example

```
public class Main
{
  int x;

  // Constructor with a parameter
  public Main(int x)
 {
    this.x = x;
 }

  // Call the constructor
  public static void main(String[] args)
 {
    Main myObj = new Main(5);
    System.out.println("Value of x = " + myObj.x);
  }
}
```

Methods to use ‘this’ in Java

Following are the ways to use the ‘this’ keyword in Java mentioned below:

- Using the ‘this’ keyword to refer to current class instance variables.
- Using this() to invoke the current class constructor
- Using ‘this’ keyword to return the current class instance
- Using ‘this’ keyword as the method parameter
- Using ‘this’ keyword to invoke the current class method
- Using ‘this’ keyword as an argument in the constructor call

1. Using 'this' keyword to refer to current class instance variables Java

```
// Java code for using 'this' keyword to
// refer current class instance variables
class Test
{
   int a;
   int b;

   // Parameterized constructor
   Test(int a, int b)
   {
      this.a = a;
      this.b = b;
   }

   void display()
   {
      // Displaying value of variables a and b
      System.out.println("a = " + a + "  b = " + b);
   }

   public static void main(String[] args)
   {
      Test object = new Test(10, 20);
      object.display();
   }
}
```

Output

```
a = 10  b = 20
```

2. Using this() to invoke current class constructor

```
// Java code for using this() to
// invoke current class constructor
class Test {
   int a;
   int b;

   // Default constructor
   Test()
   {
      this(10, 20);
      System.out.println("Inside  default constructor \n");
   }

   // Parameterized constructor
   Test(int a, int b)
   {
      this.a = a;
      this.b = b;
      System.out.println(
         "Inside parameterized constructor");
   }

   public static void main(String[] args)
   {
      Test object = new Test();
   }
}
```

Output

```
Inside parameterized constructor
Inside  default constructor
```

3. Using 'this' keyword to return the current class instance

```
// Java code for using 'this' keyword
// to return the current class instance
class Test {
   int a;
   int b;

   // Default constructor
   Test()
   {
      a = 10;
      b = 20;
   }

   // Method that returns current class instance
   Test get() { return this; }

   // Displaying value of variables a and b
   void display()
   {
      System.out.println("a = " + a + "  b = " + b);
   }

   public static void main(String[] args)
   {
      Test object = new Test();
      object.get().display();
   }
}
```

Output

```
a = 10  b = 20
```

4. Using ‘this’ keyword as a method parameter

```
// Java code for using 'this'
// keyword as method parameter
class Test {
   int a;
   int b;

   // Default constructor
   Test()
   {
      a = 10;
      b = 20;
   }

   // Method that receives 'this' keyword as parameter
   void display(Test obj)
   {
      System.out.println("a = " + obj.a
                     + "  b = " + obj.b);
   }

   // Method that returns current class instance
   void get() { display(this); }

   // main function
   public static void main(String[] args)
   {
      Test object = new Test();
      object.get();
   }
}
```

Output

```
a = 10  b = 20
```

5. Using 'this' keyword to invoke the current class method

```
// Java code for using this to invoke current
// class method
class Test {

   void display()
   {
      // calling function show()
      this.show();

      System.out.println("Inside display function");
   }

   void show()
   {
      System.out.println("Inside show function");
   }

   public static void main(String args[])
   {
      Test t1 = new Test();
      t1.display();
   }
}
```

Output

```
Inside show function
Inside display function
```

6. Using 'this' keyword as an argument in the constructor call

```
// Java code for using this as an argument in constructor call
// Class with object of Class B as its data member
class A {
   B obj;

   A(B obj)
   {
      this.obj = obj;
       // calling display method of class B
      obj.display();
   }
}
class B {
   int x = 5;

   // Default Constructor that create an object of A
   // with passing this as an argument in the
   // constructor
   B() { A obj = new A(this); }

   // method to show value of x
   void display()
   {
      System.out.println("Value of x in Class B : " + x);
   }

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

Output

```
Value of x in Class B : 5
```

Advantages of using ‘this’ reference

There are certain advantages of using ‘this’ reference in Java as mentioned below:

1. It helps to distinguish between instance variables and local variables with the same name.
2. It can be used to pass the current object as an argument to another method.
3. It can be used to return the current object from a method.
4. It can be used to invoke a constructor from another overloaded constructor in the same class.

Disadvantages of using ‘this’ reference

Although ‘this’ reference comes with many advantages there are certain disadvantages of also:

1. Overuse of this can make the code harder to read and understand.
2. Using this unnecessarily can add unnecessary overhead to the program.
3. Using this in a static context results in a compile-time error.
4. Overall, this keyword is a useful tool for working with objects in Java, but it should be used judiciously and only when necessary.

final keyword

- 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. final Variable

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

Example

```
class Main1
 {
  public static void main(String[] args)
 {
      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 message.

2. 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 message.

3.final Class

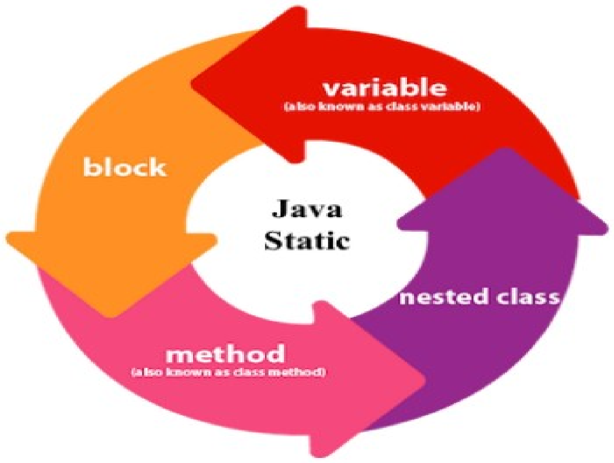
- In Java, the final class cannot be inherited by another class. For example,

```
// 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.

Static keyword

- The static keyword is a non-access modifier used for methods and attributes. Static methods/attributes can be accessed without creating an object of a class.
- The static keyword in Java is mainly used for memory management. The static keyword in Java is used to share the same variable or method of a given class.
- The users can apply static keywords with variables, methods, blocks, and nested classes. The static keyword belongs to the class than an instance of the class. The static keyword is used for a constant variable or a method that is the same for every instance of a class.
- Static can be:
  1. Blocks
  2. Variables
  3. Methods
  4. Classes



Note: To create a static member(block, variable, method, nested class), precede its declaration with the keyword static.

1.Static blocks

- If you need to do the computation in order to initialize your **static variables**, you can declare a static block that gets executed exactly once, when the class is first loaded.
- Consider the following java program demonstrating the use of static blocks.

Example:

```
// Java program to demonstrate static block

public class Main2
{
    static
    {
    System.out.println("Hi, I'm a Static Block!");
    }
  public static void main(String[] args)
    {
     //main method
     System.out.println("Hi, I'm a Main Method!");
  }
}
```

Output:

```
Hi, I'm a Static Block!

Hi, I'm a Main Method!
```

Example:

```
// Java program to demonstrate use of static blocks

class Test
{
        // static variable
        static int a = 10;
        static int b;

        // static block
        static {
                System.out.println("Static block initialized.");
                b = a * 4;
        }

        public static void main(String[] args)
        {
                System.out.println("from main");
                System.out.println("Value of a : "+a);
                System.out.println("Value of b : "+b);
        }
}
```

Output

```
Static block initialized.
from main
Value of a : 10
Value of b : 40
```

2.Static variables

- When a variable is declared as static, then a single copy of the variable is created and shared among all objects at the class level. Static variables are, essentially, global variables. All instances of the class share the same static variable.

Important points for static variables:

- We can create static variables at the class level only.
- static block and static variables are executed in the order they are present in a program.

- Below is the Java program to demonstrate that static block and static variables are executed in the order they are present in a program.

Example

```
class Test {

  // static variable
  static int max = 10;

  // non-static variable
  int min = 5;
}

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

     // access the non-static variable
     System.out.println("min + 1 = " + (obj.min + 1));

     // access the static variable
     System.out.println("max + 1 = " + (Test.max + 1));
  }
}
```

Output:

```
min + 1 = 6
max + 1 = 11
```

3.Static methods

- When a method is declared with the static keyword, it is known as the static method. The most common example of a static method is the main( ) method. As discussed above, Any static member can be accessed before any objects of its class are created, and without reference to any object. Methods declared as static have several restrictions:

  - They can only directly call other static methods.
  - They can only directly access static data.
  - They cannot refer to this or super in any way.

```
public class Main
 {

   static int age;

   static void display()
   {
      System.out.println("Static Method");
   }
   public static void main(String[] args)
  {

      // access the static variable
      age = 30;
      System.out.println("Age is " + age);

      // access the static method
      display();
   }
}
```

Output:

```
Age is 30
Static Method
```

Access Modifiers in Java

- There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.
- The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.
- There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

- There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

Understanding Java Access Modifiers

- Let's understand the access modifiers in Java by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

1) Private

- The private access modifier is accessible only within the class.

Simple example of private access modifier

- In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
class A
{
        private int data=40;
        private void msg()
        {
                System.out.println("Hello java");
        }
}
public class Simple
{
         public static void main(String args[])
        {
                A obj=new A();
                 System.out.println(obj.data);//Compile Time Error
                obj.msg();//Compile Time Error
         }
}
```

2) Default

- If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

- In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A
{
        void msg()
       {
              System.out.println("Hello");
       }
}
//save by B.java
package mypack;
import pack.*;
class B
{
        public static void main(String args[])
       {
              A obj = new A();//Compile Time Error
                obj.msg();//Compile Time Error
       }
}
```

- In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) Protected

- The protected access modifier is accessible within package and outside the package but through inheritance only.
- The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.
- It provides more accessibility than the default modifer.

Example of protected access modifier

- In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java
package pack;
public class A
{
        protected void msg()
        {
                System.out.println("Hello");
        }
}
```

```
//save by B.java
package mypack;
import pack.*;
class B extends A
{
         public static void main(String args[])
        {
                 B obj = new B();
                 obj.msg();
          }
}
```

Output:

```
Hello
```

4) Public

- The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
//save by A.java

package pack;

public class A
{
        public void msg()
        {
                System.out.println("Hello");
        }
}
```

```
//save by B.java

  package mypack;

  import pack.*;
  class B
  {
         public static void main(String args[])
        {
                A obj = new A();
                obj.msg();
        }
}
```