Introduction to Object Oriented Programming

What is OOP?

OOP stands for Object-Oriented Programming.

Before Object-Oriented Programming (OOPs), most programs used a procedural approach, where the focus was on writing step-by-step functions. This made it harder to manage and reuse code in large applications.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Object oriented Concepts

OOP concepts

1 Objects
2 Class
3 Encapsulation
4 Abstraction
5 Inheritance,
6 Polymorphism
7 message passing.

1. Class

A Class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. Using classes, you can create multiple objects with the same behavior instead of writing their code multiple times. In general, class declarations can include these components in order:

- **Modifiers**: A class can be public or have default access (Refer to this for details).
- **Class name:** The class name should begin with the initial letter capitalized by convention.
- **Body:** The class body is surrounded by braces, { }.

To create a class, use the keyword `class`:

```
public class Main {

  int x = 5;

}
```

2. Object

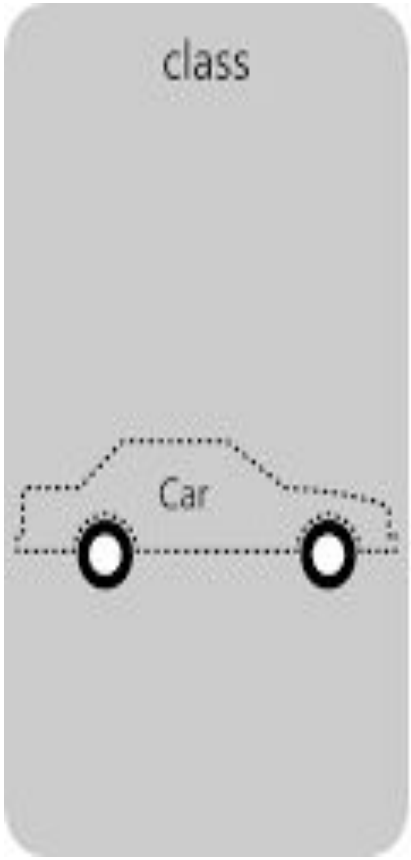
An Object is a basic unit of Object-Oriented Programming that represents real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. The objects are what perform your code, they are the part of your code visible to the viewer/user. An object mainly consists of:

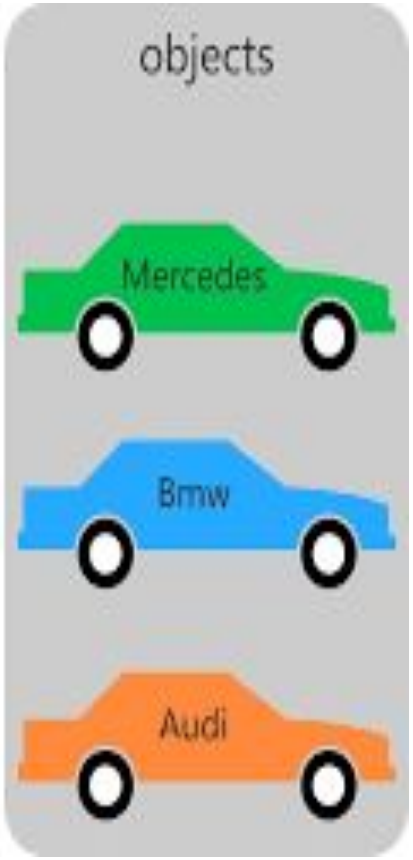
- **State**: It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior**: It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity**: It is a unique name given to an object that enables it to interact with other objects.
- **Method:** A method is a collection of statements that perform some specific task and return the result to the caller.

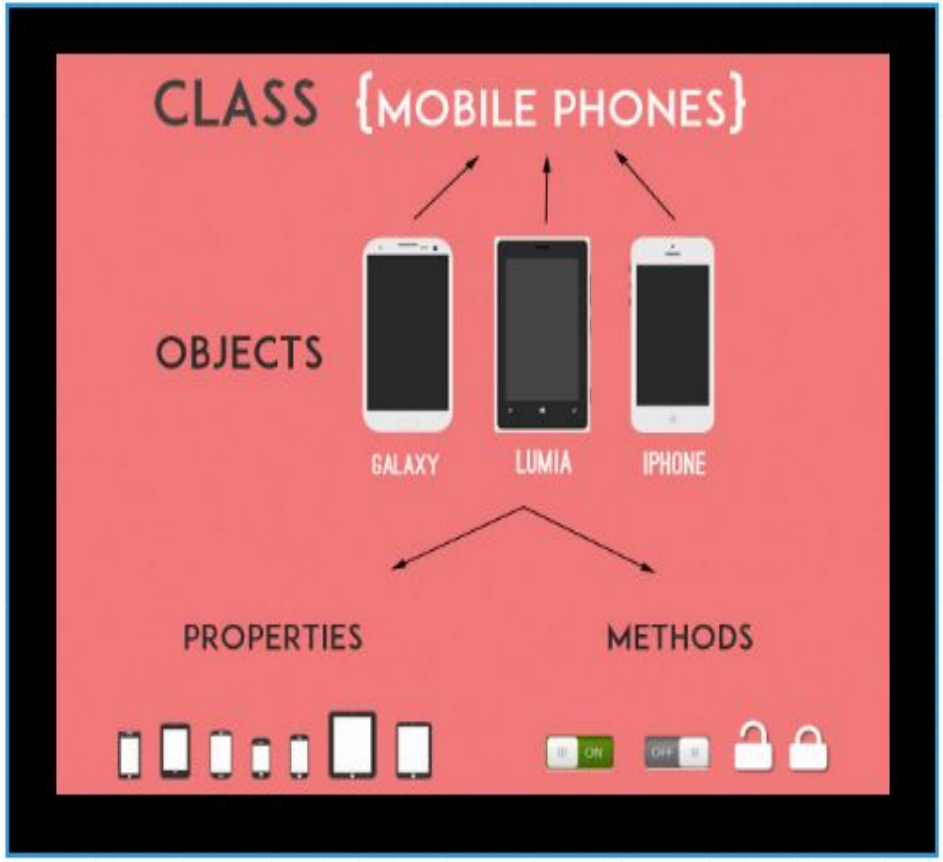
In Java, an object is created from a class. We have already created the class named `Main`, so now we can use this to create objects.

To create an object of `Main`, specify the class name, followed by the object name, and use the keyword `new`:

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

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


class
Car


objects
Mercedes
Bmw
Audi


CLASS {MOBILE PHONES}
OBJECTS
GALAXY
LUMIA
IPHONE
PROPERTIES
METHODS
ON
OFF

Abstraction

Abstraction in Java is the process of hiding the implementation details and only showing the essential details or features to the user. It allows to focus on what an object does rather than how it does it. The unnecessary details are not displayed to the user.

Note: In Java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

Encapsulation

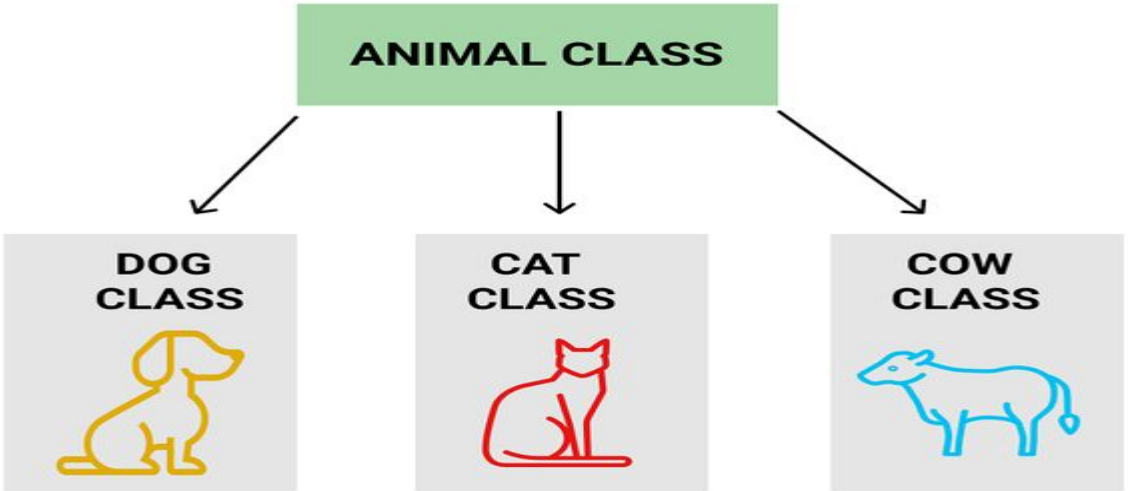
Encapsulation is defined as the process of wrapping data and the methods into a single unit, typically a class. It is the mechanism that binds together the code and the data. It manipulates. Another way to think about encapsulation is that it is a protective shield that prevents the data from being accessed by the code outside this shield.

- Technically, in encapsulation, the variables or the data in a class is hidden from any other class and can be accessed only through any member function of the class in which they are declared.
- In encapsulation, the data in a class is hidden from other classes, which is similar to what **data-hiding** does. So, the terms "encapsulation" and "data-hiding" are used interchangeably.
- Encapsulation can be achieved by declaring all the variables in a class as private and writing public methods in the class to set and get the values of the variables.

Inheritance

Inheritance is an important pillar of OOP (Object Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class. We are achieving inheritance by using **extends** keyword. Inheritance is also known as "**is-a**" relationship.

Example: Dog, Cat, Cow can be Derived Class of Animal Base Class.

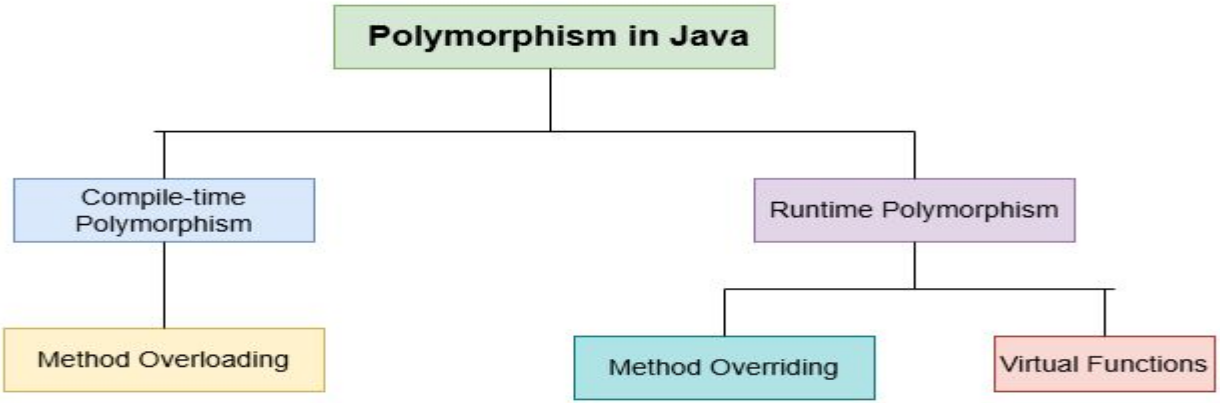


Let us discuss some frequently used important terminologies:

- **Superclass:** The class whose features are inherited is known as superclass (also known as base or parent class).
- **Subclass:** The class that inherits the other class is known as subclass (also known as derived or extended or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Polymorphism

The word **polymorphism** means having **many forms**, and it comes from the Greek words **poly (many)** and **morph (forms)**, this means one entity can take many forms. In Java, polymorphism allows the same method or object to behave differently based on the context, specially on the project's actual runtime class.



Method Overloading and Method Overriding

1. Method Overloading: Also, known as **compile-time polymorphism**, is the concept of Polymorphism where more than one method share the same name with different signature(Parameters) in a class. The return type of these methods can or cannot be same.

2. Method Overriding: Also, known as run-time polymorphism, is the concept of Polymorphism where method in the child class has the same name, return-type and parameters as in parent class. The child class provides the implementation in the method already written.

Advantage of OOPs over Procedure-Oriented Programming Language

Object-oriented programming (OOP) offers several key advantages over procedural programming:

- By using objects and classes, you can create reusable components, leading to less duplication and more efficient development.
- It provides a clear and logical structure, making the code easier to understand, maintain, and debug.
- OOP supports the DRY (Don't Repeat Yourself) principle.This principle encourages minimizing code repetition, leading to cleaner, more maintainable code. Common functionalities are placed in a single location and reused, reducing redundancy.
- By reusing existing code and creating modular components, OOP allows for quicker and more efficient application development

Disadvantages of OOP

- OOP has concepts like classes, objects, inheritance etc. For beginners, this can be confusing and takes time to learn.
- If we write a small program, using OOP can feel too heavy. We might have to write more code than needed just to follow the OOP structure.
- The code is divided into different classes and layers, so in this, finding and fixing bugs can sometimes take more time.
- OOP creates a lot of objects, so it can use more memory compared to simple programs written in a procedural way.

Java Class Attributes

we used the term "variable" for `x` in the example (as shown below). It is actually an attribute of the class. Or you could say that class attributes are variables within a class:

Create a class called "`Main`" with two attributes: `x` and `y`:

```
public class Main {
  int x = 5;
  int y = 3;
  }
```

examples:

```
public class Main {
  String fname = "John";
  String lname = "Doe";
  int age = 24;

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println("Name: " + myObj.fname + " " + myObj.lname);
    System.out.println("Age: " + myObj.age);
  }
}
```

```
Name: John Doe
Age: 24
```

Java Methods

A method is a block of code which only runs when it is called.
You can pass data, known as parameters, into a method.
Methods are used to perform certain actions, and they are also known as functions.
Why use methods? To reuse code: define the code once, and use it many times.

Create a Method

- A method must be declared within a class.
- It is defined with the name of the method, followed by parentheses ().
- Java provides some pre-defined methods, such as `System.out.println()`, but you can also create your own methods to perform certain actions:

```
public class Main {
  static void myMethod() {
    // code to be executed
  }
}
```

- `myMethod()` is the name of the method
- `static` means that the method belongs to the Main class and not an object of the Main class. You will learn more about objects and how to access methods through objects later in this tutorial.
- `void` means that this method does not have a return value. You will learn more about return values later in this chapter

Call a Method

to call a method in Java, write the method's name followed by two parentheses () and a semicolon;

```
public class Main {
  static void myMethod() {
    System.out.println("I just got executed!");
  }

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

```
I just got executed!
```

A method can also be called multiple times:

```
public class Main {
  static void myMethod() {
    System.out.println("I just got executed!");
  }

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

```
I just got executed!
I just got executed!
I just got executed!
```

Java Method Parameters

Parameters and Arguments

Information can be passed to methods as a parameter. Parameters act as variables inside the method.

Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

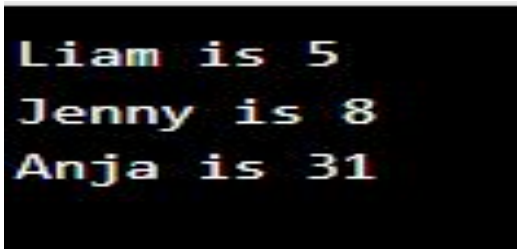
The following example has a method that takes a `String` called fname as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:


```
public class Main {
  static void myMethod(String fname) {
    System.out.println(fname + " Refsnes");
  }

  public static void main(String[] args) {
    myMethod("Liam");
    myMethod("Jenny");
    myMethod("Anja");
  }
}
```

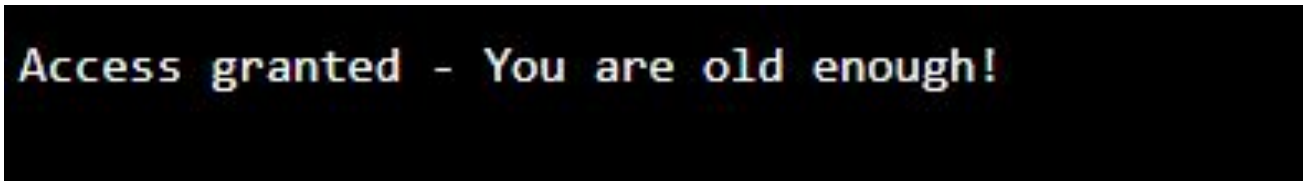
```
Liam Refsnes
Jenny Refsnes
Anja Refsnes
```

Multiple Parameters

```
public class Main {
  static void myMethod(String fname, int age) {
    System.out.println(fname + " is " + age);
  }

  public static void main(String[] args) {
    myMethod("Liam", 5);
    myMethod("Jenny", 8);
    myMethod("Anja", 31);
  }
}
```



```
public class Main {

  // Create a checkAge() method with an integer parameter called age
  static void checkAge(int age) {

    // If age is less than 18, print "access denied"
    if (age < 18) {
      System.out.println("Access denied - You are not old enough!");

    // If age is greater than, or equal to, 18, print "access granted"
    } else {
      System.out.println("Access granted - You are old enough!");
    }

  }

  public static void main(String[] args) {
    checkAge(20); // Call the checkAge method and pass along an age of 20
  }
}
```



Return Values

In the previous page, we used the `void` keyword in all examples, which indicates that the method should not return a value.
If you want the method to return a value, you can use a primitive data type (such as `int`, `char`, etc.) instead of `void`, and use the `return` keyword inside the method:

```
public class Main {
  static int myMethod(int x) {
    return 5 + x;
  }

  public static void main(String[] args) {
    System.out.println(myMethod(3));
  }
}
```

```
public class Main {
  static int myMethod(int x, int y) {
    return x + y;
  }

  public static void main(String[] args) {
    int z = myMethod(5, 3);
    System.out.println(z);
  }
}
```

Method Overloading

With method overloading, multiple methods can have the same name with different parameters:

```
int myMethod(int x)
float myMethod(float x)
  double myMethod(double x, double y)
```

Consider the following example, which has two methods that add numbers of different type:

```
public class Main {
  static int plusMethodInt(int x, int y) {
    return x + y;
  }

  static double plusMethodDouble(double x, double y) {
    return x + y;
  }

  public static void main(String[] args) {
    int myNum1 = plusMethodInt(8, 5);
    double myNum2 = plusMethodDouble(4.3, 6.26);
    System.out.println("int: " + myNum1);
    System.out.println("double: " + myNum2);
  }
}
```

```
int: 13
double: 10.559999999999999
```

Java Recursion

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.
Recursion may be a bit difficult to understand. The best way to figure out how it works is to experiment with it.

Recursion Example

Adding two numbers together is easy to do, but adding a range of numbers is more complicated. In the following example, recursion is used to add a range of numbers together by breaking it down into the simple task of adding two numbers:

```
public class Main {
  public static void main(String[] args) {
    int result = sum(10);
    System.out.println(result);
  }
  public static int sum(int k) {
    if (k > 0) {
      return k + sum(k - 1);
    } else {
      return 0;
    }
  }
}
```

```
55
```

Program For Practice

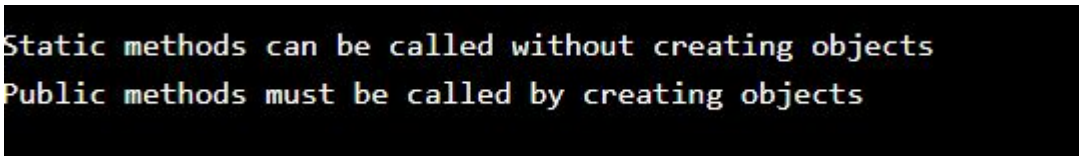
https://pwskills.com/blog/basic-java-program-examples/#Basic_Java_Program_Examples_With_Outputs

Static vs. Public

you will often see Java programs that have either `static` or `public` attributes and methods.

we created a `static` method, which means that it can be accessed without creating an object of the class, unlike `public`, which can only be accessed by objects:

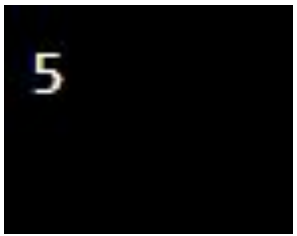
```
public class Main {
  // Static method
  static void myStaticMethod() {
    System.out.println("Static methods can be called without creating objects");
  }

  // Public method
  public void myPublicMethod() {
    System.out.println("Public methods must be called by creating objects");
  }

  // Main method
  public static void main(String[] args) {
    myStaticMethod(); // Call the static method

    Main myObj = new Main(); // Create an object of MyClass
    myObj.myPublicMethod(); // Call the public method
  }
}
```



Access Methods With an Object


```
// Create a Main class
public class Main {

  // Create a fullThrottle() method
  public void fullThrottle() {
    System.out.println("The car is going as fast as it can!");
  }

  // Create a speed() method and add a parameter
  public void speed(int maxSpeed) {
    System.out.println("Max speed is: " + maxSpeed);
  }

  // Inside main, call the methods on the myCar object
  public static void main(String[] args) {
    Main myCar = new Main();     // Create a myCar object
    myCar.fullThrottle();      // Call the fullThrottle() method
    myCar.speed(200);          // Call the speed() method
  }
}
```

```
// Create a Main class
public class Main {
  int x;

  // Create a class constructor for the Main class
  public Main() {
    x = 5;
  }

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



jdk

JDK (Java Development Kit)

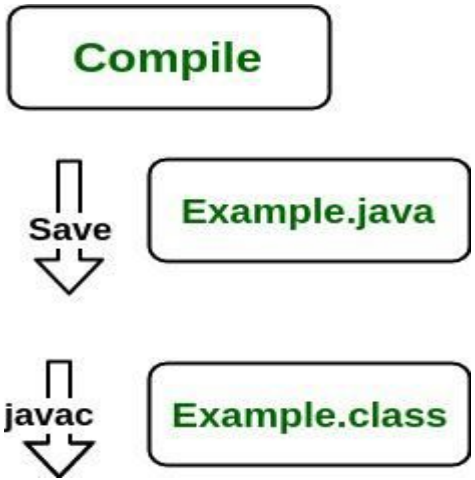
The JDK is a software development kit that provides tools to develop and run Java applications. It includes two main components:

- Development Tools (to provide an environment to develop your java programs)
- JRE (to execute your java program)

Working of JDK

The JDK enables the development and execution of Java programs. Consider the following process:

- **Java Source File (e.g., Example.java)**: You write the Java program in a source file.
- **Compilation:** The source file is compiled by the Java Compiler (part of JDK) into bytecode, which is stored in a .class file (e.g., Example.class).
- **Execution:** The bytecode is executed by the JVM (Java Virtual Machine), which interprets the bytecode and runs the Java program.



JRE ((Java Runtime Environment)

The JRE is an installation package that provides an environment to **only run(not develop)** the Java program (or application) onto your machine. JRE is only used by those who only want to run Java programs that are end-users of your system.

Working of JRE

- **Class Loader**: The JRE's class loader loads the .class file containing the bytecode into memory.
- **Bytecode Verifier**: JRE includes a bytecode verifier to ensure security before execution
- **Interpreter**: JVM uses an interpreter + JIT compiler to execute bytecode for optimal performance
- **Execution**: The program executes, making calls to the underlying hardware and system resources as needed.

JVM (Java Virtual Machine)

The **JVM** is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by line, hence it is also known as an ***interpreter.***

It is mainly responsible for three activities.

- Loading
- Linking
- Initialization

Basic Programming constructs

Refer Basics of java ppt

Bit Shift (>>, <<,>>>)

A bit shift is a Bitwise operation where the order of a series of bits is moved to efficiently perform a mathematical operation. A bit shift, shifts each digit in a number's binary representation left or right by as many spaces as specified by the second operand. These operators can be applied to integral types such as int, long, short, byte, or char.

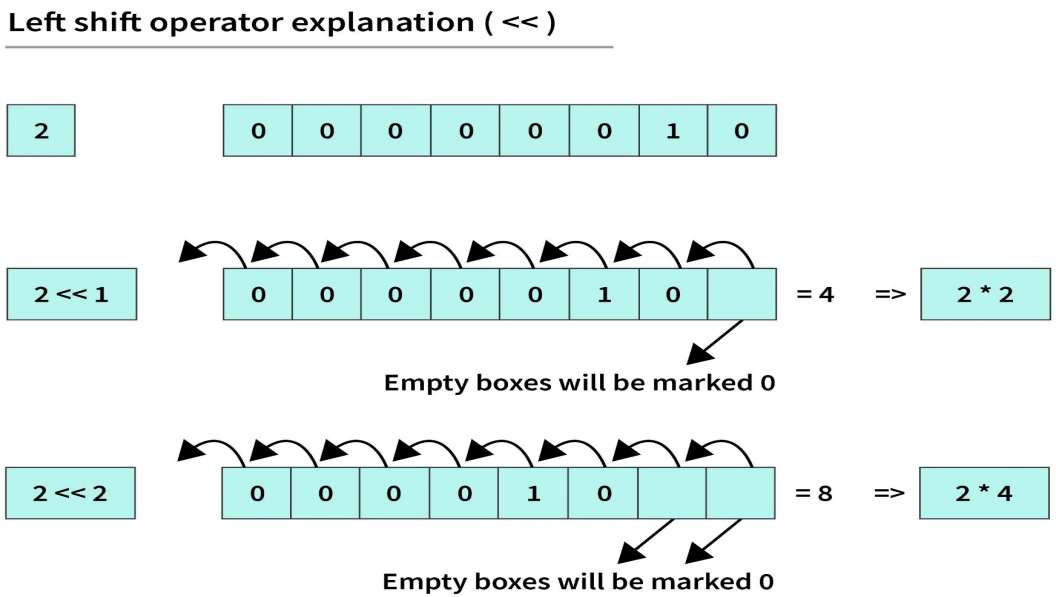
There are three types of shift:

a. Left shift: <<

b. Arithmetic/signed right shift: >>

c. Logical/unsigned right shift: >>>

a. Left shift: << is the left shift operator and meets both logical and arithmetic shifts' needs.



Arithmetic/signed right shift: >> is the arithmetic (or signed) right shift operator.

// right shift of 8

8 = 1000 (In Binary)

// perform 2 bit right shift

8 >> 2:

1000 >> 2 = 0010 (equivalent to 2)

unsigned right shift operator,

Logical/unsigned right shift: >>> is the logical (or unsigned) right shift operator. It performs same operation as right shift operator but its sign is not preserved in the operation.