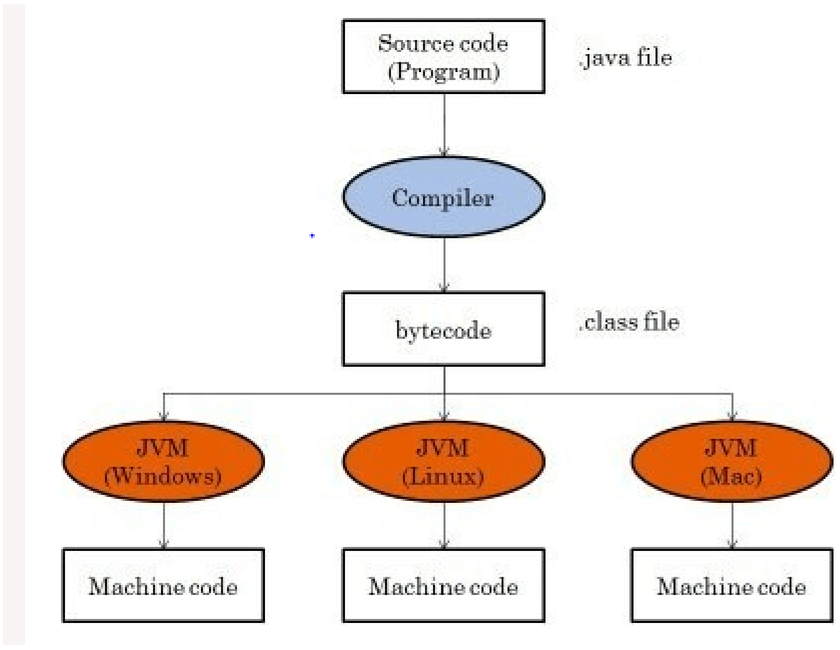
UNIT II

Byte Code

Java bytecode is the instruction set for the Java Virtual Machine. It acts similar to an assembler which is an alias representation of a C++ code. As soon as a java program is compiled, java bytecode is generated. In more apt terms, java bytecode is the machine code in the form of a .class file. With the help of java bytecode we achieve platform independence in java.



JVM (Java Virtual Machine) Architecture

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides a runtime environment in which java bytecode can be executed.

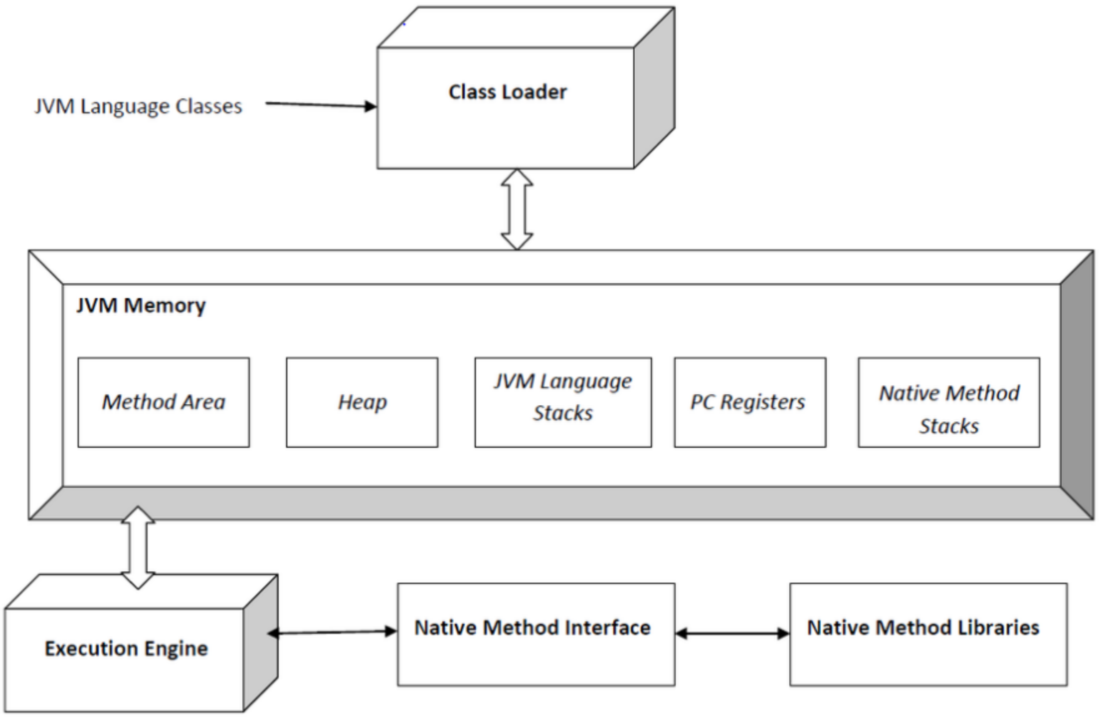
What is JVM

It is:

- **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.
- **An implementation** Its implementation is known as JRE (Java Runtime Environment).
- **Runtime Instance** Whenever you write a java command on the command prompt to run the java class, an instance of JVM is created.

JVM Architecture

Let's understand the internal architecture of JVM. It contains classloader, memory area, execution engine etc.



Class Loader Subsystem

It is mainly responsible for three activities.

- Loading
- Linking
- Initialization

Loading: The Class loader reads the "*.class*" file, generates the corresponding binary data and saves it in the method area. For each "*.class*" file, JVM stores the following information in the method area.

- The fully qualified name of the loaded class and its immediate parent class.
- Whether the "*.class*" file is related to Class or Interface or Enum.
- Modifier, Variables and Method information etc.

The JVM performs following operation:

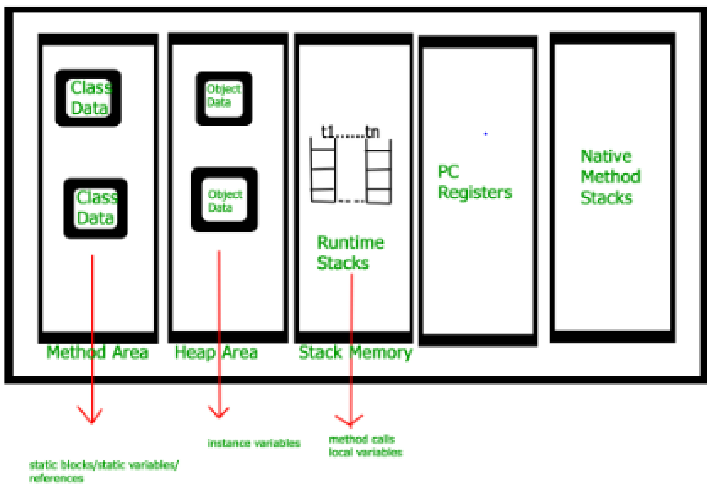
- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

JVM Memory

- **Method area:** In the method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource. From java 8, static variables are now stored in Heap area.
- **Heap area:** Information of all objects is stored in the heap area. There is also one Heap Area per JVM. It is also a shared resource.
- **Stack area:** For every thread, JVM creates one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which stores methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminates, its run-time stack will be destroyed by JVM. It is not a shared resource.
- **PC Registers:** Store address of current execution instruction of a thread. Obviously, each thread has separate PC Registers.
- **Native method stacks:** For every thread, a separate native stack is created. It stores native method information.



JDK: Java Development Kit

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop java applications and applets. It physically exists. It contains JRE + development tools.

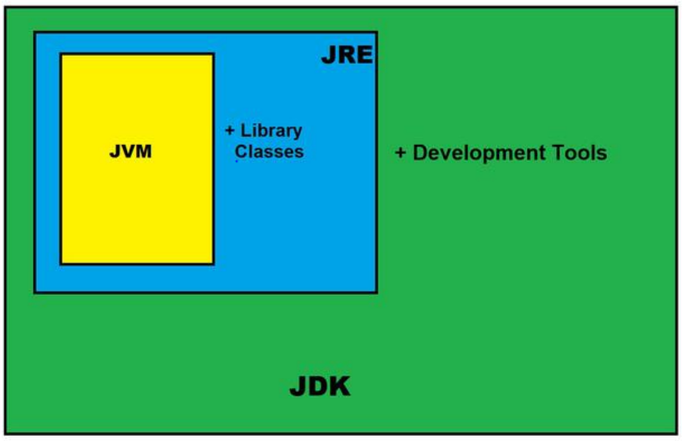
JDK is an implementation of any one of the below given Java Platforms released by Oracle corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) etc. to complete the development of a Java Application.

JDK=JRE+Development Tools

JDK Architecture:



Contents of JDK

The JDK has a private Java Virtual Machine (JVM) and a few other resources necessary for the development of a Java Application.

JDK contains:

- Java Runtime Environment (JRE),
- An interpreter/loader (Java),
- A compiler (javac),
- An archiver (jar) and many more.

The Java Runtime Environment in JDK is usually called Private Runtime because it is separated from the regular JRE and has extra content. The Private Runtime in JDK contains a JVM and all the class libraries present in the production environment, as well as additional libraries useful to developers, e.g, internationalization libraries and the IDL libraries.

Java Control Statements | Control Flow in Java

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

1. Decision Making statements
   - if statements
   - switch statement
2. Loop statements
   - do while loop
   - while loop
   - for loop
   - for-each loop
3. Jump statements
   - break statement
   - continue statement

Decision-Making statements:

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

1. Simple if statement
2. if-else statement
3. if-else-if ladder
4. Nested if-statement
5. Switch case

Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax:

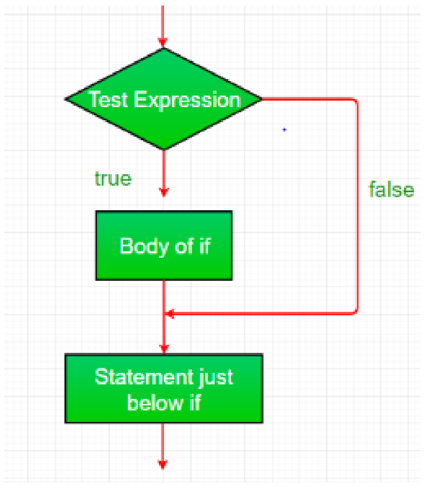
```
if(condition) {
statement 1; //executes when condition is true
}
```

Example:

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
int z=x+y;
if(x+y > 20) {
System.out.println(z+" is greater than 20");
}
}
}
```

Sample Output:

```
22 is Greater than 20
```



If-else statement

An if….else statement includes two blocks that are if block and else block. It is the next form of the control statement, which allows the execution of JavaScript in a more controlled way.

It is used when you are required to check two different conditions and execute a different set of codes. The else statement is used for specifying the execution of a block of code if the condition is false.

Syntax:

```
if (condition)
 {
  // block of code will execute if the condition is true
}
 else
 {
  // block of code will execute if the condition is false
}
```

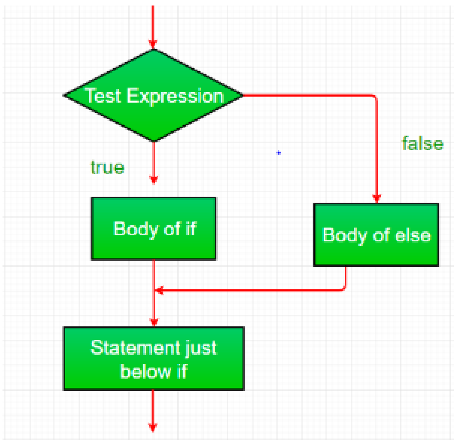
Example:

```
import java.util.*;

class IfElseDemo {
    public static void main(String args[])
    {
        int i = 10;

        if (i < 15)
            System.out.println("i is smaller than 15");
        else
            System.out.println("i is greater than 15");
    }
}
```

Sample Output:

```
            i is smaller than 15
```

Flow chart:



If-else-if ladder

Here, a user can decide among multiple options.The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that ‘if’ is executed, and the rest of the ladder is bypassed.

If none of the conditions is true, then the final else statement will be executed. There can be as many as ‘else if’ blocks associated with one ‘if’ block but only one ‘else’ block is allowed with one ‘if’ block.

Syntax:

```
if (condition1)

{

  //  block of code will execute if condition1 is true

}

 else if (condition2)

{

  //  block of code will execute if the condition1 is false and condition2 is true

}

else

{

  //  block of code will execute if the condition1 is false and condition2 is false

}
```

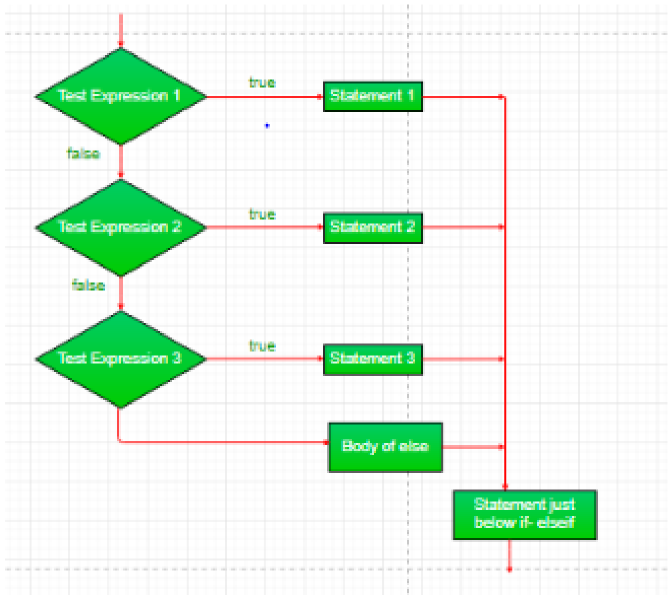
Example:

```
import java.util.*;

class ifelseifDemo {
    public static void main(String args[])
    {
        int i = 20;

        if (i == 10)
            System.out.println("i is 10");
        else if (i == 15)
            System.out.println("i is 15");
        else if (i == 20)
            System.out.println("i is 20");
        else
            System.out.println("i is not present");
    }
}
```

Sample Output:

i is 20

Flow Chart:



Nested if-statement:

A nested if is an if statement that is the target of another if or else. Nested if statements mean an if statement inside an if statement. Yes, java allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

Syntax:

```
if (condition1)
{
Statement 1; //It will execute when condition1 is true
if (condition2)
 {
Statement 2; //It will execute when condition2 is true
}
else
{
 Statement 3; //It will execute when condition2 is false
}
```

}

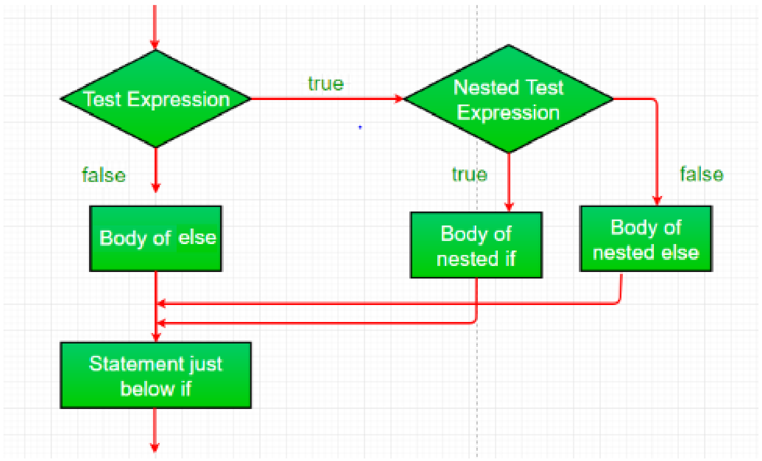
Example:

```
import java.util.*;

class NestedIfDemo {
    public static void main(String args[])
    {
        int i = 10;

        if (i == 10 || i<15) {
            // First if statement
            if (i < 15)
                System.out.println("i is smaller than 15");
            if (i < 12)
                System.out.println(
                    "i is smaller than 12 too");
        } else{
                System.out.println("i is greater than 15");
        }
    }
}
```

Sample Output:

```
i is smaller than 15
i is smaller than 12 too
```

Flow Chart:



Switch Case

The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

The **switch statement** uses the **break** or **default** keywords, but both of them are optional. Let us define these two keywords:

break: It is used within the **switch statement** for terminating the sequence of a statement. It is optional to use. If it gets omitted, then the execution will continue on each statement. When it is used, then it will stop the execution within the block.

default: It specifies some code to run when there is no case match. There can be only a single default keyword in a switch. It is also optional, but it is recommended to use it as it takes care of unexpected cases.

Syntax:

```
switch (expression)

{

 case value1:

  statement1;

  break;

 case value2:

  statement2;

  break;

 .

 .

 case valueN:

  statementN;

  break;

 default:

  statementDefault;

}
```

Example:

```
class GFG {
  public static void main (String[] args) {
    int num=20;
```

```
        switch(num){
        case 5 :  System.out.println("It is 5");
                  break;
        case 10 : System.out.println("It is 10");
                  break;
        case 15 : System.out.println("It is 15");
                  break;
        case 20 : System.out.println("It is 20");
                  break;
        default:  System.out.println("Not present");

       }
   }
}
```

Sample Output:

```
It is 20
```

Looping statement:

Looping in programming languages is a feature which facilitates the execution of a set of instructions/functions repeatedly while some condition evaluates to true. Java provides three ways for executing the loops. While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.

- do while loop
- while loop
- for loop
- for-each loop

For loop:

For loop provides a concise way of writing the loop structure. Unlike a while loop, a for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping.

- **Initialization**: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
- **Condition**: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return a boolean value either true or false. It is an optional condition.

- **Increment/Decrement**: It increments or decrements the variable value. It is an optional condition.
- **Statement**: The statement of the loop is executed each time until the second condition is false.

Syntax:

```
for(initialization; condition; increment/decrement){
//statement or code to be executed
}
```

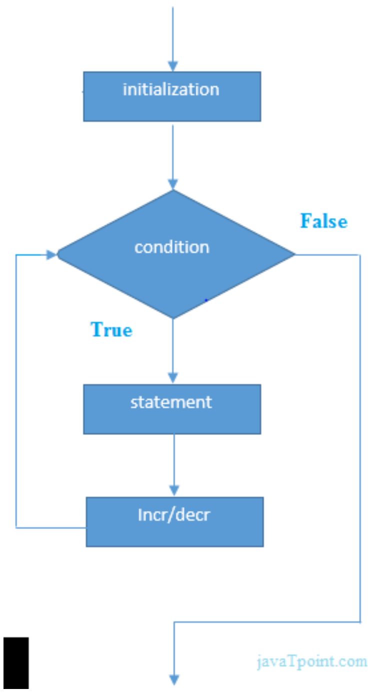
Example:

```
public class ForExample {
public static void main(String[] args) {
    //Code of Java for loop
    for(int i=1;i<=10;i++){
        System.out.println(i);
    }
}
}
```

Sample Output:

```
1
2
3
4
5
6
7
8
9
10
```

Flow Chart:



While Loop:

A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.

While loop starts with the checking of Boolean condition. If it evaluates to true, then the loop body statements are executed, otherwise the first statement following the loop is executed. For this reason it is also called **Entry control loop**

Once the condition is evaluated to true, the statements in the loop body are executed. Normally the statements contain an update value for the variable being processed for the next iteration.

Syntax:

```
while (boolean condition)
{
   loop statements...
}
```

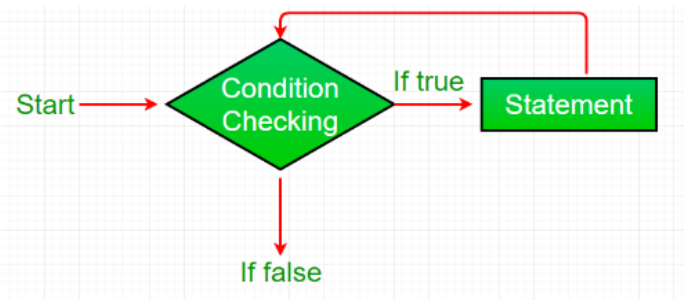
Example:

```
import java.io.*;

class GFG {
    public static void main (String[] args) {
      int i=0;
      while (i<=10)
      {
        System.out.println(i);
        i++;
      }
    }
}
```

Sample Output:

```
0
1
2
3
4
5
6
7
8
9
10
```

Flow chart:



Do…….while:

do while loop is similar to while loop with only difference that it checks for condition after executing the statements, and therefore is an example of Exit Control Loop.

do while loop starts with the execution of the statement(s). There is no checking of any condition for the first time.

After the execution of the statements, and update of the variable value, the condition is checked for true or false value. If it is evaluated to be true, the next iteration of the loop starts.

When the condition becomes false, the loop terminates which marks the end of its life cycle.

It is important to note that the do-while loop will execute its statements at least once before any condition is checked, and therefore is an example of an exit control loop.

Syntax:

```
do
{
  statements..
}
while (condition);
```

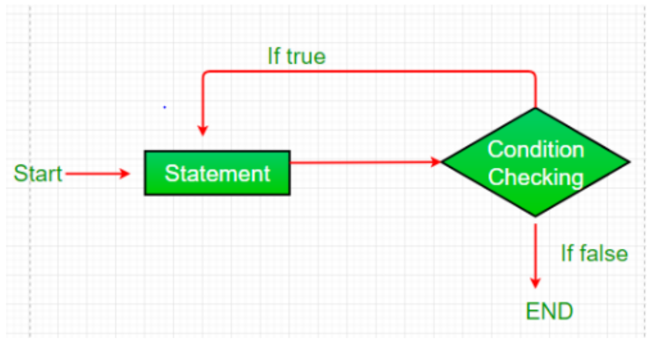
Example:

```
import java.io.*;

class GFG {
    public static void main (String[] args) {
      int i=0;
      do
      {
        System.out.println(i);
        i++;
      }while(i<=10);
    }
}
```

Sample Output:

```
0
1
2
3
4
5
6
7
8
9
10
```

Flow chart:



Jumping Statements:

Jumping statements are control statements that transfer execution control from one point to another point in the program. There are two Jump statements that are provided in the Java programming language:

1. Break statement.
2. Continue statement.

Break statement

1. Using Break Statement to exit a loop:

In java, the break statement is used to terminate the execution of the nearest looping statement or switch statement. The break statement is widely used with the switch statement, **for** loop, **while** loop, **do-while** loop.

Syntax:

```
    break;
```

Example:

```
import java.io.*;

class GFG {
   public static void main(String[] args)
   {
      int n = 10;
      for (int i = 0; i < n; i++) {
         if (i == 6)
            break;
         System.out.println(i);
      }
   }
}
```

Sample Output:

```
0
1
2
3
4
5
```

As you see, the code is meant to print 1 to 10 numbers using a for loop, but it prints only 1 to 5 . as soon as i is equal to 6, the control terminates the loop.

Continue Statements

The **continue** statement pushes the next repetition of the loop to take place, hopping any code between itself and the conditional expression that controls the loop.

Example:

```
import java.io.*;

class GFG {
    public static void main(String[] args)
    {
        for (int i = 0; i < 10; i++) {
            if (i == 6){
                 System.out.println();
                 // using continue keyword
                 // to skip the current iteration
               continue;
            }
            System.out.println(i);
        }
    }
}
```

Sample Output:

```
0
1
2
3
4
5

7
8
9
```

Class and Methods in java:

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.

A Class is like an object constructor, or a "blueprint" for creating objects.

A class in Java can contain:

- **Fields**
- **Methods**
- **Constructors**
- **Blocks**
- **Nested class and interface**

Constructors are used to create and initialize new objects in a class. Every class must have a constructor — either a default one provided by the Java compiler or a new one written for that class.

Create a Class

To create a class, use the keyword class:

Syntax:

```
          public class Main {

  int x = 5;

}
```

Create an Object

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:

Example:

```
                    public class Main {

  int x = 5;
```

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

        Main myObj = new Main();

        System.out.println(myObj.x);

      }

    }
```

Methods in java

- 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:

Example:

```
        public class Main {

  static void myMethod() {

    // code to be executed

  }

}
```

Example Explained

- 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;

In the following example, myMethod() is used to print a text (the action), when it is called:

Example:

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

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

Java Constructors

A constructor in Java is a special method that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes:

It is a special type of method which is used to initialize the object.

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 the 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

Types of Java constructors

There are two types of constructors in Java:

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

Default constructor

A constructor is similar to method and it is invoked at the time creating an object of the class, it is generally used to initialize the instance variables of a class. The constructors have the same name as their class and have no return type.

Example

```
class Bike1{

Bike1()

{System.out.println("Bike is created");}

public static void main(String args[]){

Bike1 b=new Bike1();

}

}
```

Java Parameterized Constructor

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

A parameterized constructor accepts parameters with which you can initialize the instance variables. Using a parameterized constructor, you can initialize the class variables dynamically at the time of instantiating the class with distinct values.

Syntax

```
public class Sample{

  Int i;

  public sample(int i){

    this.i = i;

  }

}
```

Example

```
public class Test {

  String val;

  Test(String val){

    this.val = val;

  }
  public static void main(String args[]){
    Test obj = new Test("test");
    System.out.println(obj.val);
  }
}
```

Overloading:

In Java, it is possible to create methods that have the same name, but different parameter lists and different definitions. This is called function overloading. Operator overloading is the ability of an operator to redefine its functionality.

In Java, "operator overloading is not supported". In method overloading, we can create methods having the same name but differ in the type, number, and/or sequence of parameters.

When an overloaded method is invoked, Java uses the type, number, and/or, sequence, or arguments as its guide to determine which version of the overloaded method to call.

Method:

If a class has multiple methods having the same name but different in parameters, it is known as Method Overloading.

If we have to perform only one operation, having the same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

Rules for Method Overloading :

- The overloaded method must change the argument list (number of parameters, data type, or sequence of parameters).
- The overloaded method can change the return type.
- The overloaded method can change the access modifier (the signature of the function should be different).

Method Overloading: changing no. of arguments

In this example, we have created two methods, the first add() method performs addition of two numbers and the second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create an instance for calling methods.

Example:

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

Sample Output:

```
22
33
```

Method Overloading: changing data type of arguments

In this example, we have created two methods that differ in data type. The first add method receives two integer arguments and the second add method receives two double arguments.

Example:

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

This Keyword

The **this** keyword refers to the current object in a method or constructor.

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). If you omit the keyword in the example above, the output would be "0" instead of "5".

this can also be used to:

- Invoke current class constructor
- Invoke current class method
- Return the current class object
- Pass an argument in the method call
- Pass an argument in the constructor call

Example:

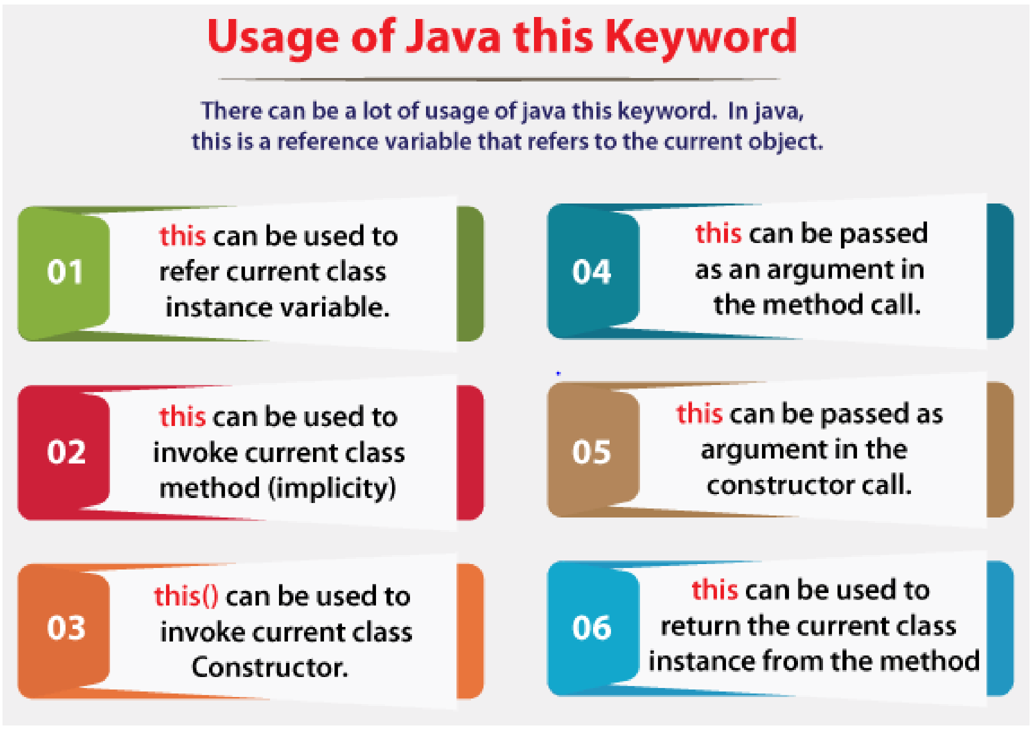
```
public class Main {

  int x;

  // Constructor with a parameter

  public Main(int x) {

    this.x = x;

  }

  public static void main(String[] args) {

    Main myObj = new Main(5);

    System.out.println("Value of x = " + myObj.x);

  }

}
```

Super Keyword in Java

The super keyword in Java is a reference variable which is used to refer to an immediate parent class object.

Whenever you create the instance of a subclass, an instance of the parent class is created implicitly which is referred to by a super reference variable.

Usage of super Keyword:


Usage of Java this Keyword
There can be a lot of usage of java this keyword. In java, this is a reference variable that refers to the current object.
01 this can be used to refer current class instance variable.
02 this can be used to invoke current class method (implicity)
03 this() can be used to invoke current class Constructor.
04 this can be passed as an argument in the method call.
05 this can be passed as argument in the constructor call.
06 this can be used to return the current class instance from the method

Static Array in Java

In Java, an array is the most important data structure that contains elements of the same type. It stores elements in contiguous memory allocation. There are two types of array i.e. **static array** and **dynamic array.** In this section, we will focus only on **static arrays in Java**.

An array that is declared with the static keyword is known as static array. It allocates memory at compile-time whose size is fixed. We cannot alter the static array.

If we want an array to be sized based on input from the user, then we cannot use static arrays. In such a case, dynamic arrays allow us to specify the size of an array at run-time.

Example:

For example, int arr[10] creates an array of size 10. It means we can insert only 10 elements; we cannot add an 11th element as the size of Array is fixed.

```
int arr[] = { 1, 3, 4 }; // static integer array
int* arr = new int[3]; // dynamic integer array
```

Static Array Java Program

```
public class StaticArrayExample
{
private static String[] array;
static
{
array = new String[2];
array[0] = "Welcome to";
array[1] = "Javatpoint";
}
public static void main(String args[])
{
for(int i = 0; i < array.length; i++)
{
System.out.print(array[i] + " ");
}
}
}
```

Sample Output:

```
Welcome to Javatpoint
```