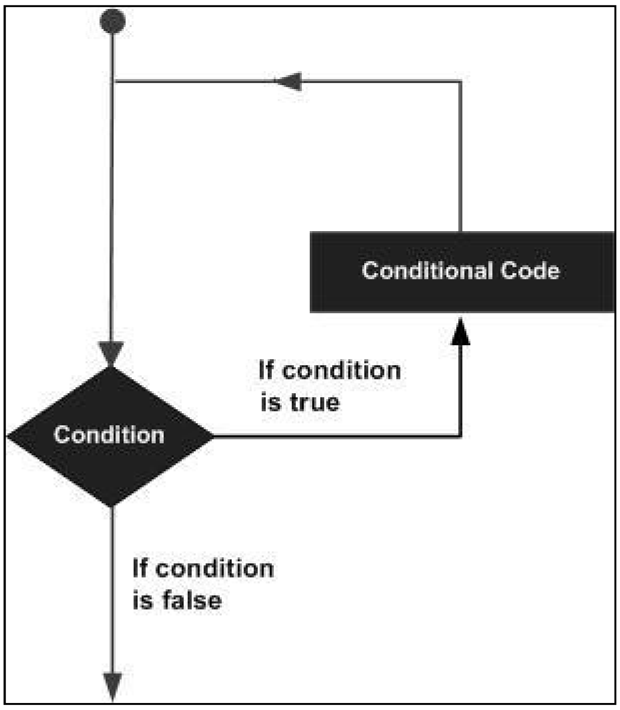
1 JAVA – LOOP CONTROL

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A **loop** statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:



Java programming language provides the following types of loop to handle looping requirements. Click the following links to check their detail.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while loop	Like a while statement, except that it tests the condition at the end of the loop body.

1.1 WHILE LOOP IN JAVA

A **while** loop statement in Java programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

The syntax of a while loop is:

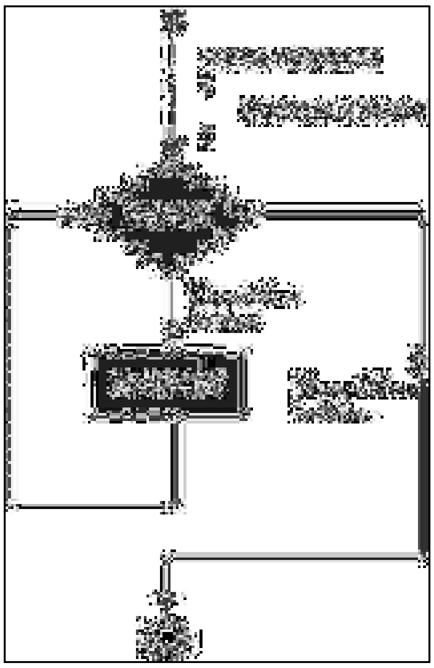
```
while(Boolean_expression)
{
   //Statements
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value.

When executing, if the *boolean_expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

When the condition becomes false, program control passes to the line immediately following the loop.

Flow Diagram



Here, key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
public class Test {

  public static void main(String args[]) {
    int x = 10;

    while (x < 20) {
      System.out.print("value of x : " + x);
      x++;
      System.out.print("\n");
    }
  }
}
```

This will produce the following result:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

1.2 FOR LOOP IN JAVA

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times.

A **for** loop is useful when you know how many times a task is to be repeated.

Syntax

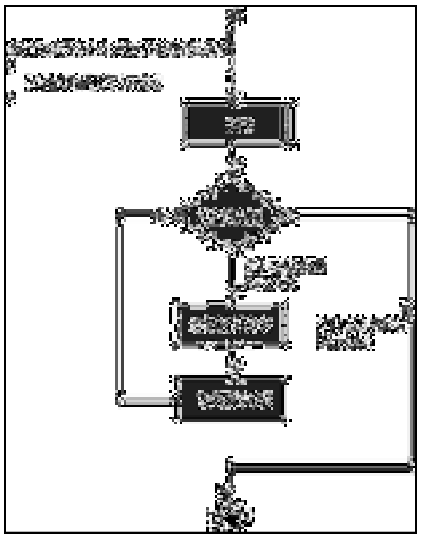
The syntax of a for loop is:

```
for(initialization; Boolean_expression; update)
{
   //Statements
}
```

Here is the flow of control in a **for** loop:

- The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables and this step ends with a semi colon (;).
- Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop will not be executed and control jumps to the next statement past the for loop.
- After the body of the for loop gets executed, the control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank with a semicolon at the end.
- The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

Flow Diagram



Example

Following is an example code of the for loop in Java.

```
public class Test {
  public static void main(String args[]) {
    for (int x = 10; x < 20; x = x + 1) {
      System.out.print("value of x : " + x);
```

```
         System.out.print("\n");
      }
   }
}
```

This will produce the following result:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

1.3 Do While Loop in Java

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax

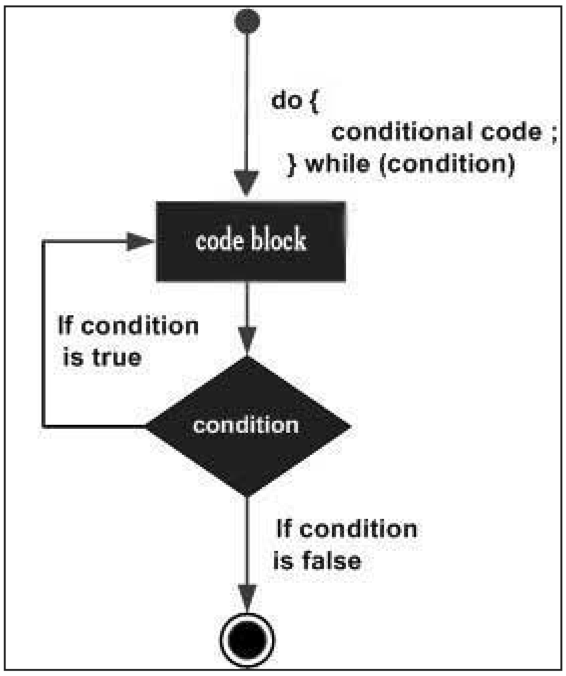
Following is the syntax of a do...while loop:

```
do
{
   //Statements
}while(Boolean_expression);
```

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

If the Boolean expression is true, the control jumps back up to do statement, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

Flow Diagram



Example

```
public class Test {
  public static void main(String args[]) {
    int x = 10;

    do {
      System.out.print("value of x : " + x);
      x++;
      System.out.print("\n");
    } while (x < 20);
  }
}
```

This will produce the following result:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

1.4 Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Java supports the following control statements. Click the following links to check their detail.

Control Statement	Description
break statement	Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

1.5 Break Statement in Java

The **break** statement in Java programming language has the following two usages:

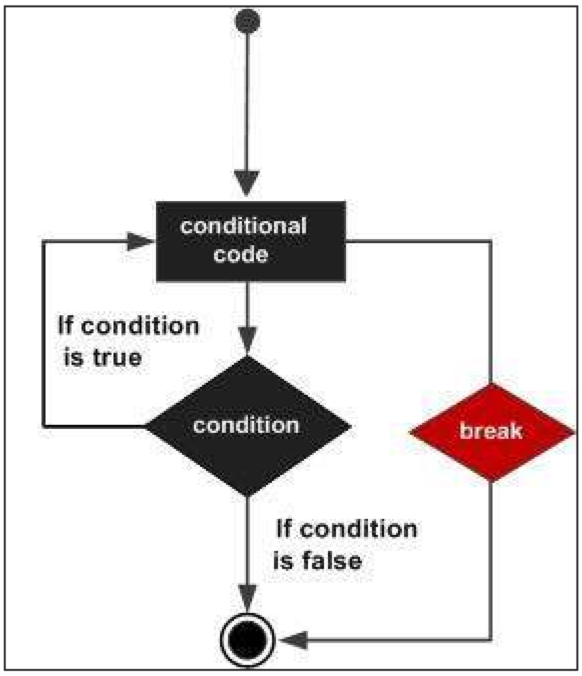
- When the break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- It can be used to terminate a case in the **switch** statement (covered in the next chapter).

Syntax

The syntax of a break is a single statement inside any loop:

```
break;
```

Flow Diagram



Example

```
public class Test {
  public static void main(String args[]) {
    int[] numbers = {10, 20, 30, 40, 50};
    for (int x : numbers) {
      if (x == 30) {
        break;
      }
      System.out.print(x);
      System.out.print("\n");
    }
  }
}
```

This will produce the following result:

```
10

20
```

1.6 Continue Statement in Java

The **continue** keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

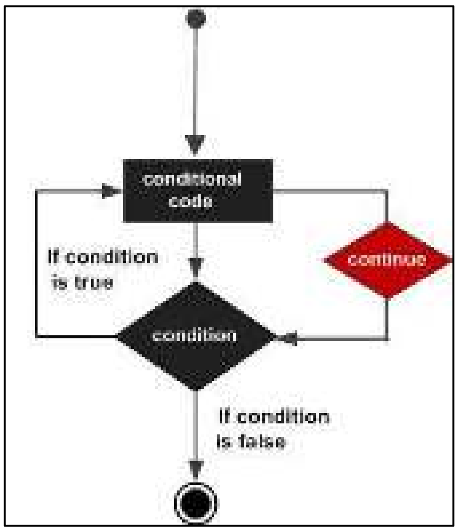
- In a for loop, the continue keyword causes control to immediately jump to the update statement.
- In a while loop or do/while loop, control immediately jumps to the Boolean expression.

Syntax

The syntax of a continue is a single statement inside any loop:

```
continue;
```

Flow Diagram



Example

```
public class Test {
  public static void main(String args[]) {
    int[] numbers = {10, 20, 30, 40, 50};
    for (int x : numbers) {
      if (x == 30) {
        continue;
      }
      System.out.print(x);
      System.out.print("\n");
    }
  }
}
```

This will produce the following result:

```
10

20

40

50
```

1.7 Enhanced for Loop in Java

As of Java 5, the enhanced for loop was introduced. This is mainly used to traverse collection of elements including arrays.

Syntax

Following is the syntax of enhanced for loop:

```
for(declaration : expression)
{
   //Statements
}
```

- Declaration: The newly declared block variable, is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
- Expression: This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

Example

```java
public class Test {

  public static void main(String args[]) {
    int[] numbers = {10, 20, 30, 40, 50};

    for (int x : numbers) {
      System.out.print(x);
      System.out.print(",");
    }
    System.out.print("\n");
    String[] names = {"James", "Larry", "Tom", "Lacy"};
    for (String name : names) {
      System.out.print(name);
      System.out.print(",");
    }
  }
}
```

This will produce the following result:

```
10,20,30,40,50,

James,Larry,Tom,Lacy,
```

2 JAVA – ARRAYS

Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

This tutorial introduces how to declare array variables, create arrays, and process arrays using indexed variables.

2.1 DECLARING ARRAY VARIABLES

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable:

```
dataType[] arrayRefVar; // preferred way.

or

dataType arrayRefVar[];  // works but not preferred way.
```

Note: The style **dataType[] arrayRefVar** is preferred. The style **dataType arrayRefVar[]** comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

Example

The following code snippets are examples of this syntax:

```
double[] myList; // preferred way.

or

double myList[]; //  works but not preferred way.
```

2.2 CREATING ARRAYS

You can create an array by using the new operator with the following syntax:

```
arrayRefVar = new dataType[arraySize];
```

The above statement does two things:

- It creates an array using new dataType[arraySize].
- It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below:

```
dataType[] arrayRefVar = new dataType[arraySize];
```

Alternatively, you can create arrays as follows:

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

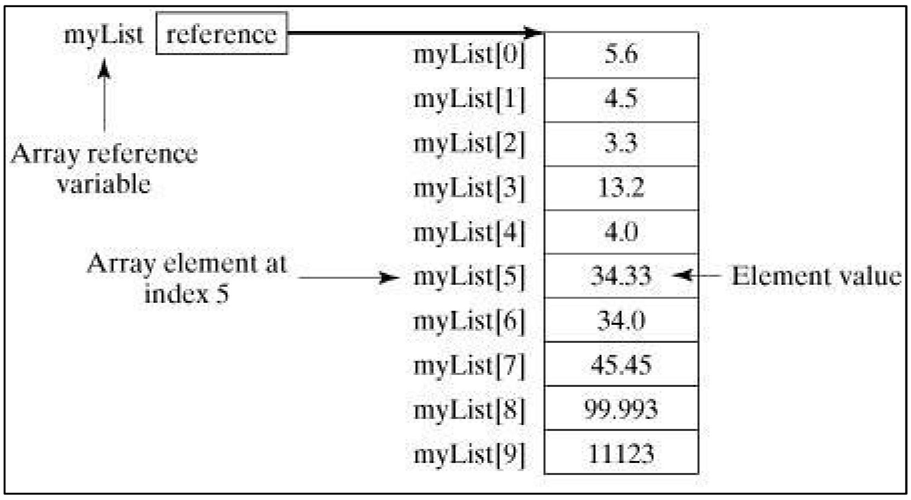
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Example

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList:

```
double[] myList = new double[10];
```

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



2.3 Processing Arrays

When processing array elements, we often use either **for** loop or **foreach** loop because all of the elements in an array are of the same type and the size of the array is known.

Example

Here is a complete example showing how to create, initialize, and process arrays:

```
public class TestArray {

  public static void main(String[] args) {
    double[] myList = {1.9, 2.9, 3.4, 3.5};

    // Print all the array elements
    for (int i = 0; i < myList.length; i++) {
      System.out.println(myList[i] + " ");
    }
    // Summing all elements
    double total = 0;
    for (int i = 0; i < myList.length; i++) {
      total += myList[i];
    }
    System.out.println("Total is " + total);
    // Finding the largest element
    double max = myList[0];
    for (int i = 1; i < myList.length; i++) {
      if (myList[i] > max) max = myList[i];
    }
    System.out.println("Max is " + max);
  }
}
```

This will produce the following result:

```
1.9

2.9

3.4

3.5

Total is 11.7

Max is 3.5
```

2.4 The foreach Loops

JDK 1.5 introduced a new for loop known as foreach loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

Example

The following code displays all the elements in the array myList:

```
public class TestArray {

  public static void main(String[] args) {
    double[] myList = {1.9, 2.9, 3.4, 3.5};

    // Print all the array elements
    for (double element : myList) {
      System.out.println(element);
    }
  }
}
```

This will produce the following result:

```
1.9

2.9

3.4

3.5
```

2.5 Passing Arrays to Methods

Just as you can pass primitive type values to methods, you can also pass arrays to methods. For example, the following method displays the elements in an **int** array:

```
public static void printArray(int[] array)
  { for (int i = 0; i < array.length; i++)
  {

    System.out.print(array[i] + " ");
  }
}
```

You can invoke it by passing an array. For example, the following statement invokes the printArray method to display 3, 1, 2, 6, 4, and 2:

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

2.6 Returning an Array from a Method

A method may also return an array. For example, the following method returns an array that is the reversal of another array:

```
public static int[] reverse(int[] list) {
  int[] result = new int[list.length];


  for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {
    result[j] = list[i];
  }
  return result;
```

2.7 The Arrays Class

The java.util.Arrays class contains various static methods for sorting and searching arrays, comparing arrays, and filling array elements. These methods are overloaded for all primitive types.

Sr. No.	Methods with Description
1	**public static int binarySearch(Object[] a, Object key)** Searches the specified array of Object (Byte, Int , double, etc.) for the specified value using the binary search algorithm. The array must be sorted prior to making this call. This returns index of the search key, if it is contained in the list; otherwise, it returns (–(insertion point + 1)).
2	**public static boolean equals(long[] a, long[] a2)** Returns true if the two specified arrays of longs are equal to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. This returns true if the two arrays are equal. Same method could be used by all other primitive data types (Byte, short, Int, etc.)

3	**public static void fill(int[] a, int val)** **Assigns the specified int value to each element of the specified array of ints. The same method could be used by all other primitive data types (Byte, short, Int, etc.)**
4	**public static void sort(Object[] a)** **Sorts the specified array of objects into an ascending order, according to the natural ordering of its elements. The same method could be used by all other primitive data types (Byte, short, Int, etc.)**

3 JAVA – METHODS

A Java method is a collection of statements that are grouped together to perform an operation. When you call the System.out.**println()** method, for example, the system actually executes several statements in order to display a message on the console.

Now you will learn how to create your own methods with or without return values, invoke a method with or without parameters, and apply method abstraction in the program design.

3.1 CREATING METHOD

Considering the following example to explain the syntax of a method:

```
public static int methodName(int a, int b) {
  // body
}
```

Here,

- public static: modifier
- int: return type
- methodName: name of the method
- a, b: formal parameters
- int a, int b: list of parameters

Method definition consists of a method header and a method body. The same is shown in the following syntax:

```
modifier returnType nameOfMethod (Parameter List) {
 // method body
}
```

The syntax shown above includes:

- modifier: It defines the access type of the method and it is optional to use.
- returnType: Method may return a value.
- nameOfMethod: This is the method name. The method signature consists of the method name and the parameter list.

- Parameter List: The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- method body: The method body defines what the method does with the statements.

Example

Here is the source code of the above defined method called **max()**. This method takes two parameters num1 and num2 and returns the maximum between the two:

```
/** the snippet returns the minimum between two numbers */
public static int minFunction(int n1, int n2) {

   int min;

   if (n1 > n2)
      min = n2;

   else

      min = n1;

   return min;

}
```

3.2 Method Calling

For using a method, it should be called. There are two ways in which a method is called i.e., method returns a value or returning nothing (no return value).

The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when:

- the return statement is executed.
- it reaches the method ending closing brace.

The methods returning void is considered as call to a statement. Let's consider an example:

```
System.out.println("This is example.com!");
```

The method returning value can be understood by the following example:

```
int result = sum(6, 9);
```

Example

Following is the example to demonstrate how to define a method and how to call it:

```
public class ExampleMinNumber {

  public static void main(String[] args) {
    int a = 11;
    int b = 6;
    int c = minFunction(a, b);
    System.out.println("Minimum Value = " + c);
  }

  /**
   * returns the minimum of two numbers
   */
  public static int minFunction(int n1, int n2) {
    int min;
    if (n1 > n2) min = n2;
    else
      min = n1;
    return min;
  }
}
```

This will produce the following result:

```
Minimum value = 6
```

3.3 THE VOID KEYWORD

The void keyword allows us to create methods which do not return a value. Here, in the following example we're considering a void method *methodRankPoints*. This method is a void method, which does not return any value. Call to a void method must be a statement *methodRankPoints(255.7);*. It is a Java statement which ends with a semicolon as shown in the following example.

Example

```
public class ExampleVoid {
  public static void main(String[] args) {
    methodRankPoints(255.7);
  }

  public static void methodRankPoints(double points) {
    if (points >= 202.5) {
      System.out.println("Rank:A1");
    } else if (points >= 122.4) {
      System.out.println("Rank:A2");
    } else {
      System.out.println("Rank:A3");
    }
  }
}
```

This will produce the following result:

```
Rank:A1
```

3.4 Passing Parameters by Value

While working under calling process, arguments is to be passed. These should be in the same order as their respective parameters in the method specification. Parameters can be passed by value or by reference.

Passing Parameters by Value means calling a method with a parameter. Through this, the argument value is passed to the parameter.

Example

The following program shows an example of passing parameter by value. The values of the arguments remain the same even after the method invocation.

```java
public class swappingExample {

  public static void main(String[] args) {
    int a = 30;
    int b = 45;

    System.out.println("Before swapping, a = " +
        a + " and b = " + b);

    // Invoke the swap method swapFunction(a, b);
    System.out.println("\n**Now, Before and After swapping values will
be same here**:");
    System.out.println("After swapping, a = " +
        a + " and b is " + b);
  }

  public static void swapFunction(int a, int b) {

    System.out.println("Before swapping(Inside), a = " + a
        + " b = " + b);
    // Swap n1 with n2
    int c = a;
    a = b;
    b = c;
    System.out.println("After swapping(Inside), a = " + a
        + " b = " + b);
  }
}
```

This will produce the following result:

```
Before swapping, a = 30 and b = 45
Before swapping(Inside), a = 30 b = 45
After swapping(Inside), a = 45 b = 30


**Now, Before and After swapping values will be same here**:
After swapping, a = 30 and b is 45
```

3.5 Method Overloading

When a class has two or more methods by the same name but different parameters, it is known as method overloading. It is different from overriding. In overriding, a method has the same method name, type, number of parameters, etc.

Let's consider the example discussed earlier for finding minimum numbers of integer type. If, let's say we want to find the minimum number of double type. Then the concept of overloading will be introduced to create two or more methods with the same name but different parameters.

The following example explains the same:

```
public class ExampleOverloading {

  public static void main(String[] args) {
    int a = 11;
    int b = 6;
    double c = 7.3;
    double d = 9.4;
    int result1 = minFunction(a, b);
    // same function name with different parameters
    double result2 = minFunction(c, d);
    System.out.println("Minimum Value = " + result1);
    System.out.println("Minimum Value = " + result2);
  }

  // for integer
  public static int minFunction(int n1, int n2) {
    int min;
    if (n1 > n2) min = n2;
    else
      min = n1;
    return min;
  }

  // for double
  public static double minFunction(double n1, double n2) {
    double min;
    if (n1 > n2) min = n2;
    else
      min = n1;
    return min;
  }
}
```

This will produce the following result:

```
Minimum Value = 6
Minimum Value = 7.3
```

Overloading methods makes program readable. Here, two methods are given by the same name but with different parameters. The minimum number from integer and double types is the result.

3.6 Using Command-Line Arguments

Sometimes you will want to pass some information into a program when you run it. This is accomplished by passing command-line arguments to main( ).

A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. They are stored as strings in the String array passed to main( ).

Example

The following program displays all of the command-line arguments that it is called with:

```
public class CommandLine {

  public static void main(String args[]) {
    for (int i = 0; i < args.length; i++) {
      System.out.println("args[" + i + "]: " + args[i]);
    }
  }
}
```

Try executing this program as shown here:

```
$java CommandLine this is a command line 200 -100
```

This will produce the following result:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: command
args[4]: line
args[5]: 200

args[6]: -100
```

3.7 The Constructors

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

Example

Here is a simple example that uses a constructor without parameters:

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

  // Following is the constructor
  MyClass() {
    x = 10;
  }
}
```

You will have to call constructor to initialize objects as follows:

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

3.8 Parameterized Constructor

Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

Example

Here is a simple example that uses a constructor with a parameter:

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

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

You will need to call a constructor to initialize objects as follows:

```java
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);
  }
}
```

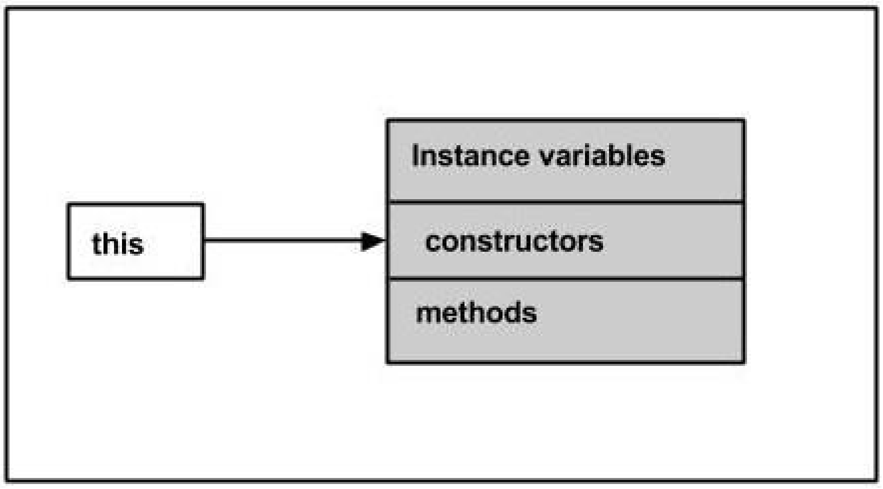
This will produce the following result:

```
10 20
```

3.9 The this keyword

this is a keyword in Java which is used as a reference to the object of the current class, with in an instance method or a constructor. Using *this* you can refer the members of a class such as constructors, variables and methods.

Note: The keyword *this* is used only within instance methods or constructors



In general, the keyword *this* is used to:

- Differentiate the instance variables from local variables if they have same names, within a constructor or a method.

```
• class Student {
    int age;

    Student(int age) {
      this.age = age;
    }
  }
```

- Call one type of constructor (parametrized constructor or default) from other in a class. It is known as explicit constructor invocation.

```
• class Student {
    int age

    Student() {
      this(20);
    }

    Student(int age) {
      this.age = age;
    }
  }
```

Example

Here is an example that uses *this* keyword to access the members of a class. Copy and paste the following program in a file with the name, **This_Example.java**.

```
public class This_Example {
   //Instance variable num
   int num = 10;
```

```
  This_Example() {
    System.out.println("This is an example program on keyword this ");
  }

  This_Example(int num) {
    //Invoking the default constructor
    this();
    //Assigning the local variable num to the instance variable num
    this.num = num;
  }

  public static void main(String[] args) {
    //Instantiating the class
    This_Example obj1 = new This_Example();
    //Invoking the print method
    obj1.print();
    //Passing a new value to the num variable through parametrized
constructor
    This_Example obj2 = new This_Example(30);
    //Invoking the print method again
    obj2.print();
  }

  public void greet() {
    System.out.println("Hi Welcome to Example");
  }

  public void print() {
    //Local variable num
    int num = 20;
    //Printing the instance variable
    System.out.println("value of local variable num is : " + num);
    //Printing the local variable
    System.out.println("value of instance variable num is : " +
this.num);
    //Invoking the greet method of a class this.greet();
  }
}
```

This will produce the following result:

```
This is an example program on keyword this
value of local variable num is : 20

value of instance variable num is : 10
Hi Welcome to Example

This is an example program on keyword this
value of local variable num is : 20

value of instance variable num is : 30
Hi Welcome to Example
```

3.10 VARIABLE ARGUMENTS(VAR-ARGS)

JDK 1.5 enables you to pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows:

```
typeName... parameterName
```

In the method declaration, you specify the type followed by an ellipsis (...). Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

Example

```java
public class VarargsDemo {
  public static void main(String args[]) {
    // Call method with variable args
    printMax(34, 3, 3, 2, 56.5);
    printMax(new double[]{1, 2, 3});
  }

  public static void printMax(double... numbers) {
    if (numbers.length == 0) {
      System.out.println("No argument passed");
      return;
    }
    double result = numbers[0];
    for (int i = 1; i < numbers.length; i++)
      if (numbers[i] > result)
        result = numbers[i];
    System.out.println("The max value is " + result);
  }
}
```

This will produce the following result:

```
The max value is 56.5

The max value is 3.0
```

3.11 THE FINALIZE( ) METHOD

It is possible to define a method that will be called just before an object's final destruction by the garbage collector. This method is called **finalize( )**, and it can be used to ensure that an object terminates cleanly.

For example, you might use finalize( ) to make sure that an open file owned by that object is closed.

To add a finalizer to a class, you simply define the finalize( ) method. The Java runtime calls that method whenever it is about to recycle an object of that class.

Inside the finalize( ) method, you will specify those actions that must be performed before an object is destroyed.

The finalize( ) method has this general form:

```
protected void finalize( )
{
   // finalization code here
}
```

Here, the keyword protected is a specifier that prevents access to finalize( ) by code defined outside its class.

This means that you cannot know when or even if finalize( ) will be executed. For example, if your program ends before garbage collection occurs, finalize( ) will not execute.

4 JAVA - OBJECT ORIENTED

Java is an Object-Oriented Language. As a language that has the Object-Oriented feature, Java supports the following fundamental concepts:

- Polymorphism
- Inheritance
- Encapsulation
- Abstraction
- Classes
- Objects
- Instance

In this chapter, we will look into the concepts - Classes and Objects.

- Object - Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.
- Class - A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

4.1 OBJECTS IN JAVA

Let us now look deep into what are objects. If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior.

If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging the tail, running.

If you compare the software object with a real-world object, they have very similar characteristics.

Software objects also have a state and a behavior. A software object's state is stored in fields and behavior is shown via methods.

So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

4.2 Classes in Java

A class is a blueprint from which individual objects are created. Following is a sample of a class.

```
public class Dog {
  String breed;
  int ageC
  String color;

  void barking() {
  }

  void hungry() {
  }

  void sleeping() {
  }
}
```

A class can contain any of the following variable types.

- **Local variables:** Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables:** Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Class variables:** Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.

Following are some of the important topics that need to be discussed when looking into classes of the Java Language.

4.3 Constructors

When discussing about classes, one of the most important sub topic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class.

Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

Following is an example of a constructor:

```
public class Puppy {
  public Puppy() {
  }

  public Puppy(String name) {
// This constructor has one parameter, name.
  }
}
```

Java also supports Singleton Classes where you would be able to create only one instance of a class.

Note: We have two different types of constructors. We are going to discuss constructors in detail in the subsequent chapters.

4.4 How to Use Singleton Class?

The Singleton's purpose is to control object creation, limiting the number of objects to only one. Since there is only one Singleton instance, any instance fields of a Singleton will occur only once per class, just like static fields. Singletons often control access to resources, such as database connections or sockets.

For example, if you have a license for only one connection for your database or your JDBC driver has trouble with multithreading, the Singleton makes sure that only one connection is made or that only one thread can access the connection at a time.

4.5 Implementing Singletons

4.5.1 Example 1

The easiest implementation consists of a private constructor and a field to hold its result, and a static accessor method with a name like getInstance().

The private field can be assigned from within a static initializer block or, more simply, using an initializer. The getInstance( ) method (which must be public) then simply returns this instance –

```
// File Name: Singleton.java
public class Singleton {
  private static Singleton singleton = new Singleton();

  /* A private Constructor prevents any other
  * class from instantiating.
  */
  private Singleton() {
  }
  /* Static 'instance' method */
  public static Singleton getInstance() {
    return singleton;
  }

  /* Other methods protected by singleton-ness */
  protected static void demoMethod() {
    System.out.println("demoMethod for singleton");
  }
}
```

Here is the main program file, where we will create a singleton object:

```
// File Name: SingletonDemo.java
public class SingletonDemo {
  public static void main(String[] args) {
    Singleton tmp = Singleton.getInstance();
    tmp.demoMethod();
  }
}
```

This will produce the following result –

```
demoMethod for singleton
```

4.5.2 Example 2

Following implementation shows a classic Singleton design pattern:

```
public class ClassicSingleton {

  private static ClassicSingleton instance = null;

  private ClassicSingleton() {
  // Exists only to defeat instantiation.
  }

  public static ClassicSingleton getInstance() {
    if (instance == null) {
      instance = new ClassicSingleton();
    }
    return instance;
  }
}
```

The ClassicSingleton class maintains a static reference to the lone singleton instance and returns that reference from the static getInstance() method.

Here, ClassicSingleton class employs a technique known as lazy instantiation to create the singleton; as a result, the singleton instance is not created until the getInstance() method is called for the first time. This technique ensures that singleton instances are created only when needed.

4.6 Creating an Object

As mentioned previously, a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the new keyword is used to create new objects.

There are three steps when creating an object from a class:

- **Declaration:** A variable declaration with a variable name with an object type.
- **Instantiation:** The 'new' keyword is used to create the object.
- **Initialization:** The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Following is an example of creating an object:

```
public class Puppy {

  public Puppy(String name) {
    // This constructor has one parameter, name.
    System.out.println("Passed Name is :" + name);
  }

  public static void main(String[] args) {
    // Following statement would create an object myPuppy
    Puppy myPuppy = new Puppy("tommy");
  }
}
```

If we compile and run the above program, then it will produce the following result:

```
Passed Name is :tommy
```

4.7 Accessing Instance Variables and Methods

Instance variables and methods are accessed via created objects. To access an instance variable, following is the fully qualified path:

```
/* First create an object */
ObjectReference = new Constructor();
/* Now call a variable as follows */
ObjectReference.variableName;
/* Now you can call a class method as follows */
ObjectReference.MethodName();
```

4.7.1.1 *Example*

This example explains how to access instance variables and methods of a class.

```java
public class Puppy {

  int puppyAge;

  public Puppy(String name) {
    // This constructor has one parameter, name.
    System.out.println("Name chosen is :" + name);
  }

  public static void main(String[] args) {
    /* Object creation */
    Puppy myPuppy = new Puppy("tommy");
    /* Call class method to set puppy's age */
    myPuppy.setAge(2);
    /* Call another class method to get puppy's age */
    myPuppy.getAge();
    /* You can access instance variable as follows as well */
    System.out.println("Variable Value :" + myPuppy.puppyAge);
  }

  public int getAge() {
    System.out.println("Puppy's age is :" + puppyAge);
    return puppyAge;
  }

  public void setAge(int age) {
    puppyAge = age;
  }
}
```

If we compile and run the above program, then it will produce the following result:

```
Name chosen is :tommy
Puppy's age is :2
Variable Value :2
```

4.8 Source File Declaration Rules

As the last part of this section, let's now look into the source file declaration rules. These rules are essential when declaring classes, *import* statements and *package* statements in a source file.

- There can be only one public class per source file.
- A source file can have multiple non-public classes.
- The public class name should be the name of the source file as well which should be appended by .java at the end. For example: the class name is *public class Employee{}* then the source file should be as Employee.java.

- If the class is defined inside a package, then the package statement should be the first statement in the source file.
- If import statements are present, then they must be written between the package statement and the class declaration. If there are no package statements, then the import statement should be the first line in the source file.
- Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file.

Classes have several access levels and there are different types of classes; abstract classes, final classes, etc. We will be explaining about all these in the access modifiers chapter.

Apart from the above mentioned types of classes, Java also has some special classes called Inner classes and Anonymous classes.

4.9 Java Package

In simple words, it is a way of categorizing the classes and interfaces. When developing applications in Java, hundreds of classes and interfaces will be written, therefore categorizing these classes is a must as well as makes life much easier.

4.10 Import Statements

In Java if a fully qualified name, which includes the package and the class name is given, then the compiler can easily locate the source code or classes. Import statement is a way of giving the proper location for the compiler to find that particular class.

For example, the following line would ask the compiler to load all the classes available in directory java_installation/java/io:

```
import java.io.*;
```

4.11 A Simple Case Study

For our case study, we will be creating two classes. They are Employee and EmployeeTest.

First open notepad and add the following code. Remember this is the Employee class and the class is a public class. Now, save this source file with the name Employee.java.

The Employee class has four instance variables - name, age, designation and salary. The class has one explicitly defined constructor, which takes a parameter.

```
import java.io.*;
```

```java
public class Employee {

  String name;
  int age;
  String designation;
  double salary;

  // This is the constructor of the class Employee
  public Employee(String name) {
    this.name = name;
  }

  // Assign the age of the Employee to the variable age.
  public void empAge(int empAge) {
    age = empAge;
  }

  /* Assign the designation to the variable designation.*/
  public void empDesignation(String empDesig) {
    designation = empDesig;
  }

  /* Assign the salary to the variable salary.*/
  public void empSalary(double empSalary) {
    salary = empSalary;
  }

  /* Print the Employee details */
  public void printEmployee() {
    System.out.println("Name:" + name);
    System.out.println("Age:" + age);
    System.out.println("Designation:" + designation);
    System.out.println("Salary:" + salary);
  }
}
```

As mentioned previously in this tutorial, processing starts from the main method. Therefore, in order for us to run this Employee class there should be a main method and objects should be created. We will be creating a separate class for these tasks.

Following is the *EmployeeTest* class, which creates two instances of the class Employee and invokes the methods for each object to assign values for each variable.

Save the following code in EmployeeTest.java file.

```java
import java.io.*;

public class EmployeeTest {
  public static void main(String args[]) {
    /* Create two objects using constructor */
    Employee empOne = new Employee("James Smith");
    Employee empTwo = new Employee("Mary Anne");

    // Invoking methods for each object created
    empOne.empAge(26);
    empOne.empDesignation("Senior Software Engineer");
    empOne.empSalary(1000);
    empOne.printEmployee();
    empTwo.empAge(21);
    empTwo.empDesignation("Software Engineer");
    empTwo.empSalary(500);
    empTwo.printEmployee();
  }
}
```

Now, compile both the classes and then run *EmployeeTest* to see the result as follows:

```
C:\> javac Employee.java
C:\> javac EmployeeTest.java
C:\> java EmployeeTest
Name:James Smith

Age:26
Designation:Senior Software Engineer
Salary:1000.0

Name:Mary Anne
Age:21

Designation:Software Engineer
Salary:500.0
```

4.12 Java – Inheritance

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance, the information is made manageable in a hierarchical order.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

4.12.1 extends Keyword

extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

```
class Super{
   .....
   .....
}
class Sub extends Super{
   .....
   .....
}
```

4.12.2 Sample Code

Following is an example demonstrating Java inheritance. In this example, you can observe two classes namely Calculation and My_Calculation.

Using extends keyword, the My_Calculation inherits the methods addition() and Subtraction() of Calculation class.

Copy and paste the following program in a file with name My_Calculation.java

```
class Calculation {
  int z;

  public void addition(int x, int y) {
    z = x + y;
    System.out.println("The sum of the given numbers:" + z);
  }

  public void Substraction(int x, int y) {
    z = x - y;
    System.out.println("The difference between the given numbers:" + z);
  }
}

public class My_Calculation extends Calculation {
  public static void main(String args[]) {
    int a = 20, b = 10;
    My_Calculation demo = new My_Calculation();
    demo.addition(a, b);
    demo.Substraction(a, b);
    demo.multiplication(a, b);
  }

  public void multiplication(int x, int y) {
    z = x * y;
    System.out.println("The product of the given numbers:" + z);
  }
}
```

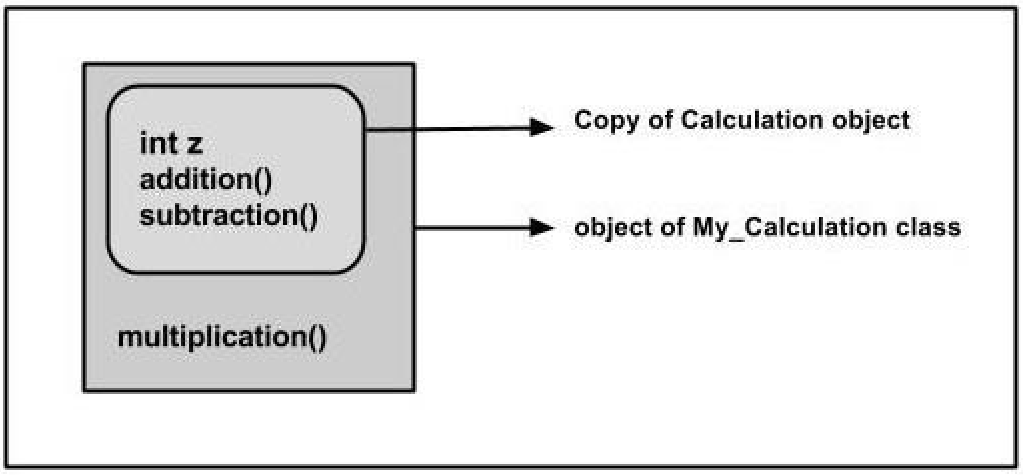
Compile and execute the above code as shown below.

```
javac My_Calculation.java
java My_Calculation
```

After executing the program, it will produce the following result.

```
The sum of the given numbers:30
The difference between the given numbers:10
The product of the given numbers:200
```

In the given program, when an object to **My_Calculation** class is created, a copy of the contents of the superclass is made within it. That is why, using the object of the subclass you can access the members of a superclass.



The Superclass reference variable can hold the subclass object, but using that variable you can access only the members of the superclass, so to access the members of both classes it is recommended to always create reference variable to the subclass.

If you consider the above program, you can instantiate the class as given below. But using the superclass reference variable (Cal in this case) you cannot call the method **multiplication()**, which belongs to the subclass My_Calculation.

```
Calculation cal = new My_Calculation();
demo.addition(a, b);
demo.Subtraction(a, b);
```

Note − A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

4.12.3 The super keyword

The **super** keyword is similar to **this** keyword. Following are the scenarios where the super keyword is used.

- It is used to differentiate the members of superclass from the members of subclass, if they have same names.
- It is used to **invoke the superclass** constructor from subclass.

4.12.4 Differentiating the Members

If a class is inheriting the properties of another class. And if the members of the superclass have the names same as the sub class, to differentiate these variables we use super keyword as shown below.

```
super.variable
super.method();
```

4.12.4.1 Sample Code

This section provides you a program that demonstrates the usage of the **super** keyword.

In the given program, you have two classes namely *Sub_class* and *Super_class*, both have a method named display() with different implementations, and a variable named num with different values. We are invoking display() method of both classes and printing the value of the variable num of both classes. Here you can observe that we have used super keyword to differentiate the members of superclass from subclass.

Copy and paste the program in a file with name Sub_class.java.

```
class Super_class {

  int num = 20;
  //display method of superclass
  public void display() {
    System.out.println("This is the display method of superclass");
  }
}


public class Sub_class extends Super_class {
  int num = 10;

  public static void main(String args[]) {
    Sub_class obj = new Sub_class();
    obj.my_method();

  }
  //display method of sub class
  public void display() {
    System.out.println("This is the display method of subclass");
  }

  public void my_method() {

    //Instantiating subclass
    Sub_class sub = new Sub_class();
    //Invoking the display() method of sub class
    sub.display();

    //Invoking the display() method of superclass
    super.display();

    //printing the value of variable num of subclass
    System.out.println("value of the variable named num in sub class:" +
sub.num);

    //printing the value of variable num of superclass
    System.out.println("value of the variable named num in super class:"
+ super.num);
   }
}
```

Compile and execute the above code using the following syntax.

```
javac Super_Demo
java Super
```

On executing the program, you will get the following result –

```
This is the display method of subclass
This is the display method of superclass

value of the variable named num in sub class:10
value of the variable named num in super class:20
```

4.12.5 Invoking Superclass Constructor

If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the superclass. But if you want to call a parameterized constructor of the superclass, you need to use the super keyword as shown below.

```
super(values);
```

4.12.5.1 Sample Code

The program given in this section demonstrates how to use the super keyword to invoke the parametrized constructor of the superclass. This program contains a superclass and a subclass, where the superclass contains a parameterized constructor which accepts a string value, and we used the super keyword to invoke the parameterized constructor of the superclass.

Copy and paste the following program in a file with the name Subclass.java

```
class Superclass {
  int age;

  Superclass(int age) {
    this.age = age;
  }

  public void getAge() {
    System.out.println("The value of the variable named age in super
class is: " + age);
  }
}

public class Subclass extends Superclass {
  Subclass(int age) {
    super(age);
  }

  public static void main(String argd[]) {
    Subclass s = new Subclass(24);
    s.getAge();
  }
}
```

Compile and execute the above code using the following syntax.

```
javac Subclass
java Subclass
```

On executing the program, you will get the following result –

```
The value of the variable named age in super class is: 24
```

4.12.6 IS-ARelationship

IS-A is a way of saying: This object is a type of that object. Let us see how the **extends** keyword is used to achieve inheritance.

```
public class Animal{
}
public class Mammal extends Animal{
}
public class Reptile extends Animal{
}
public class Dog extends Mammal{
}
```

Now, based on the above example, in Object-Oriented terms, the following are true –

- Animal is the superclass of Mammal class.

- Animal is the superclass of Reptile class.
- Mammal and Reptile are subclasses of Animal class.
- Dog is the subclass of both Mammal and Animal classes.

Now, if we consider the IS-A relationship, we can say –

- Mammal IS-A Animal
- Reptile IS-A Animal
- Dog IS-A Mammal
- Hence: Dog IS-A Animal as well

With the use of the extends keyword, the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass.

We can assure that Mammal is actually an Animal with the use of the instance operator.

Example

```
class Animal {
}

class Mammal extends Animal {
}

class Reptile extends Animal {
}

public class Dog extends Mammal {

  public static void main(String args[]) {
    Animal a = new Animal();
    Mammal m = new Mammal();
    Dog d = new Dog();
    System.out.println(m instanceof Animal);
    System.out.println(d instanceof Mammal);
    System.out.println(d instanceof Animal);
  }
}
```

This will produce the following result –

```
true
true
true
```

Since we have a good understanding of the **extends** keyword, let us look into how the **implements** keyword is used to get the IS-A relationship.

Generally, the **implements** keyword is used with classes to inherit the properties of an interface. Interfaces can never be extended by a class.

Example

```
public interface Animal {
}

public class Mammal implements Animal {
}

public class Dog extends Mammal {
}
```

4.12.7 The instanceof Keyword

Let us use the **instanceof** operator to check determine whether Mammal is actually an Animal, and dog is actually an Animal.

```
interface Animal{}

class Mammal implements Animal{}

public class Dog extends Mammal{}
```

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

    Mammal m = new Mammal();
    Dog d = new Dog();
    System.out.println(m instanceof Animal);
    System.out.println(d instanceof Mammal);
    System.out.println(d instanceof Animal);
  }
}
```

This will produce the following result:

```
true
true
true
```

4.12.8 HAS-Arelationship

These relationships are mainly based on the usage. This determines whether a certain class **HAS-A** certain thing. This relationship helps to reduce duplication of code as well as bugs.

Let's look into an example –

```
public class Vehicle {
}
```

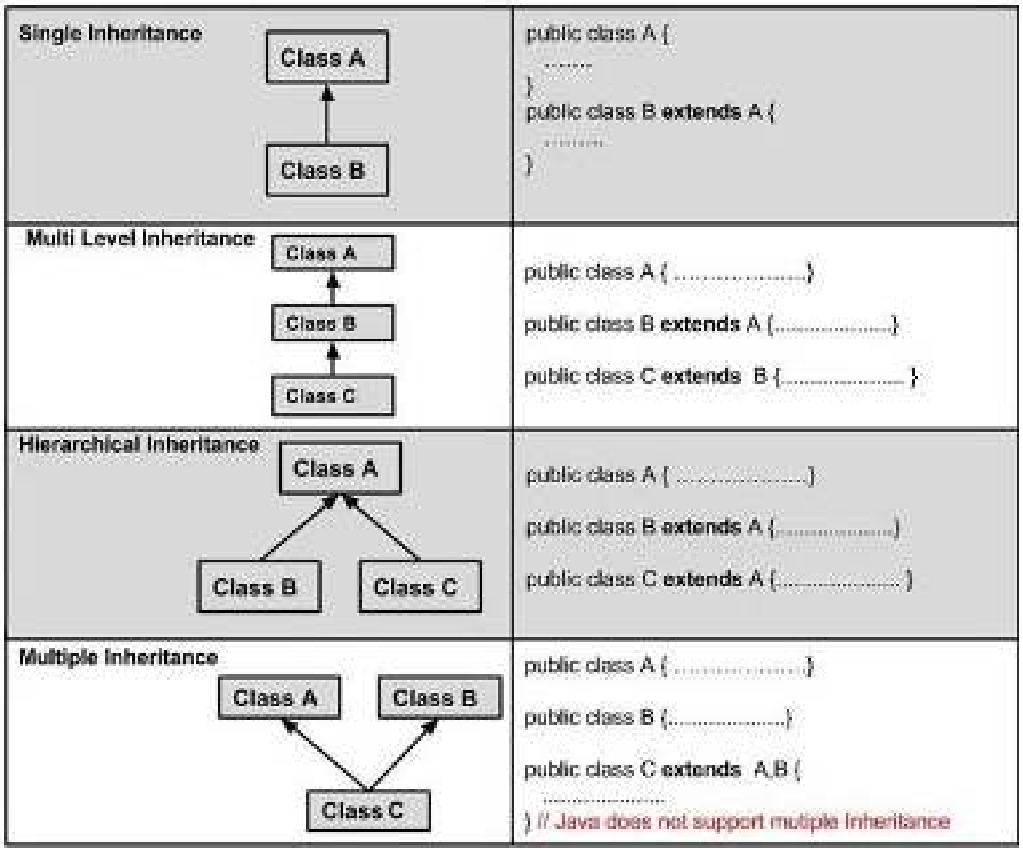
```java
public class Speed {
}

public class Van extends Vehicle {
  private Speed sp;
}
```

This shows that class Van HAS-A Speed. By having a separate class for Speed, we do not have to put the entire code that belongs to speed inside the Van class, which makes it possible to reuse the Speed class in multiple applications.

In Object-Oriented feature, the users do not need to bother about which object is doing the real work. To achieve this, the Van class hides the implementation details from the users of the Van class. So, basically what happens is the users would ask the Van class to do a certain action and the Van class will either do the work by itself or ask another class to perform the action.

4.12.9 Types of Inheritance

There are various types of inheritance as demonstrated below.



```
public class A {
  .......
}
public class B extends A {
  .........
}
```

```
public class A {....................}

public class B extends A {....................}

public class C extends  B {.................... }
```

```
public class A { ....................}

public class B extends A {....................}

public class C extends A {....................}
```

```
public class A {....................}

public class B {....................}

public class C extends  A,B {
  ....................
} // Java does not support multiple Inheritance
```

A very important fact to remember is that Java does not support multiple inheritance. This means that a class cannot extend more than one class. Therefore, following is illegal −

```
public class extends Animal, Mammal{}
```

However, a class can implement one or more interfaces, which had helped Java get rid of the impossibility of multiple inheritance.