This module was prepared by the Faculty of Computing and Software Engineering in Arba Minch University for exit exam tutorial purpose only.

ARBA MINCH UNIVERSTY

INSTITUTE OF TECHNOLOGY

FACULTY OF COMPUTING AND SOFTWARE ENGINEERING

Object Oriented Programming Course Module

(SENG 2081)

Prepared by:

Behayilu Metiku (MSc)

Arba Minch, Ethiopia

March 2023

Table of Contents

CHAPTER ONE

1. INTRODUCTION TO OBJECT ORIENTED PROGRAMMING

Object-Oriented Programming (OOP) is a programming language model organized around objects rather than actions and data. An object-oriented program can be characterized as data controlling access to code. Concepts of OOPS

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object

Object means a real word entity such as pen, chair, table etc. Any entity that has state and behavior isknown as an object. Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing details of each other's data or code, the only necessary thing is that the type of message accepted and type of response returned by the objects.

An object has three characteristics:

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

Class

Collection of objects is called class. It is a logical entity. A class can also be defined as a blueprint from which you can create an individual object. A class consists of Data members and methods. The primarypurpose of a class is to hold data/information. The member functions determine the behavior of the class,

i.e. provide a definition for supporting various operations on data held in the form of an object. Class doesn't store any space.

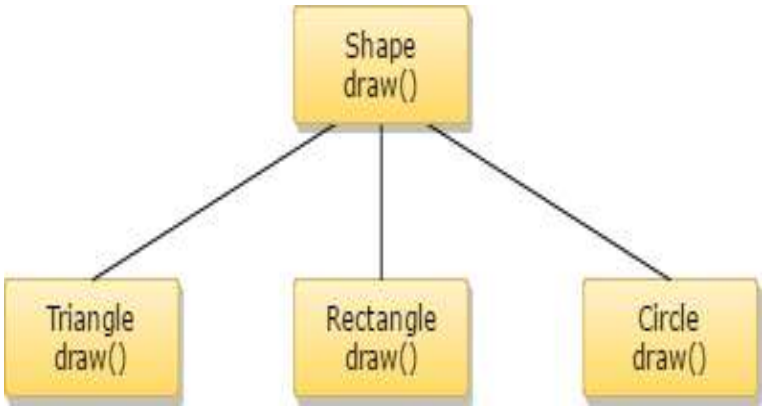
Inheritance

Inheritance can be defined as the procedure or mechanism of acquiring all the properties andbehavior of one class to another, i.e., acquiring the properties and behavior of child class from the parent class. When

one object acquires all the properties and behaviors of another object, it is known as inheritance. It provides code reusability and establishes relationships between different classes. A class which inherits the properties is known as Child Class(sub-class or derived class) whereas a class whose properties are inherited is known as Parent class(super-class or base class). Types of inheritance in java: single, multilevel and hierarchical inheritance. Multiple and hybrid inheritance is supported through interface only.

Polymorphism

When one task is performed by different ways i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.



Polymorphism is classified into two ways:

Method Overloading(Compile time Polymorphism)

Method Overloading is a feature that allows a class to have two or more methods having the same name but the arguments passed to the methods are different. Compile time polymorphism refers to a process in which a call to an overloaded method is resolved at compile time rather than at run time.

Method Overriding(Run time Polymorphism)

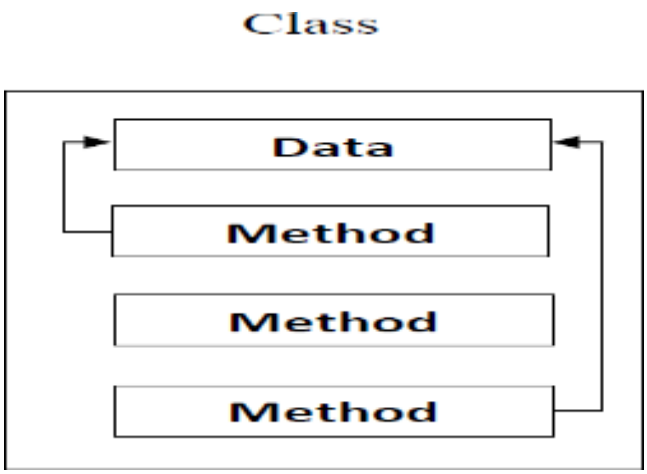
If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in java. In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

Abstraction

Abstraction is a process of hiding the implementation details and showing only functionality to the user. For example: phone call, we don't know the internal processing. In java, we use abstract class and interface to achieve abstraction.

Encapsulation

Encapsulation in java is a process of wrapping code and data together into a single unit, for example capsule i.e. mixed of several medicines. A java class is the example of encapsulation.



1.1. Difference Between Procedure-Oriented And Object-OrientedProgramming

Procedure-Oriented Programming	Object-Oriented Programming
In POP, program is divided into small parts called **functions**	In OOP, program is divided into parts called **objects**.
In POP,Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function,it can be kept public or private so we can control the access of data.
POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

1.2. Features Of Java

The main objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some awesome features which play important role in the popularity of this language. The features of Java are also known as java *buzzwords*.

A list of most important features of Java language are given below.

Simple

Java is very easy to learn and its syntax is simple, clean and easy to understand. According to Sun, Java language is a simple programming language because:

- Java syntax is based on C++ (so easier for programmers to learn it after C++).
- Java has removed many confusing and rarely-used features e.g. explicit pointers, operator overloading etc.
- There is no need to remove unreferenced objects because there is Automatic Garbage Collection in java.

Object-oriented

Java is object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behaviour. Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

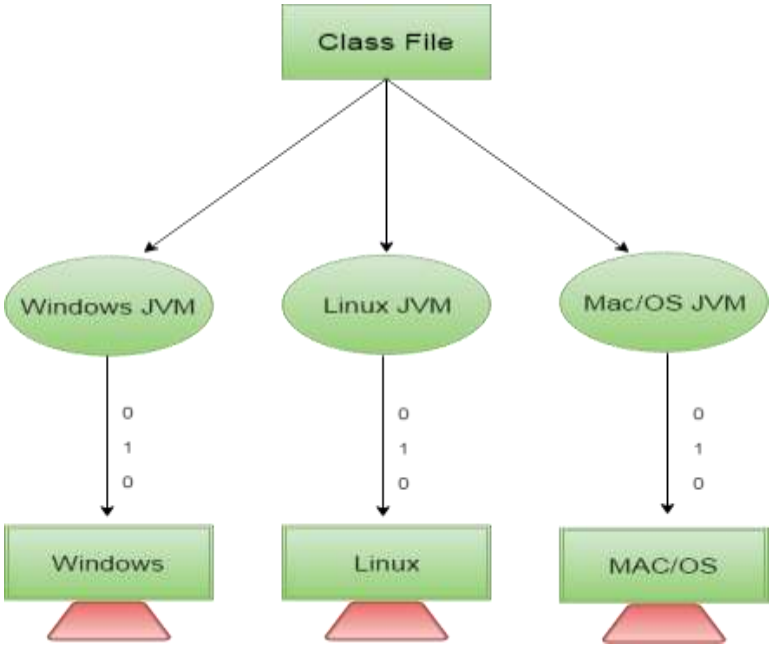
Basic concepts of OOPs are:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

Platform Independent

Java is platform independent because it is different from other languages like C, C++ etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.

There are two types of platforms software-based and hardware-based. Java provides software-based platform.



The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:

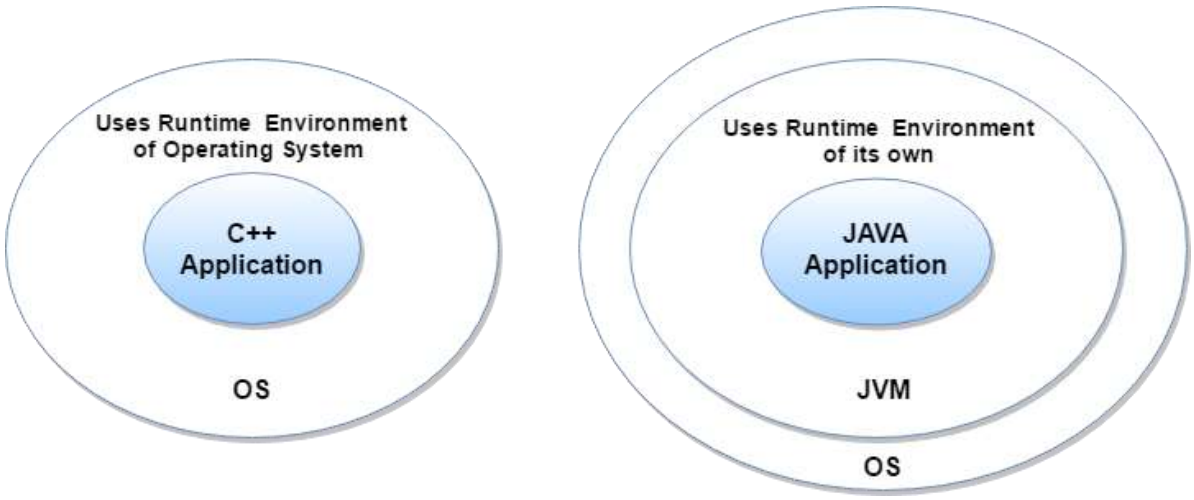
1. Runtime Environment
2. API(Application Programming Interface)

Java code can be run on multiple platforms e.g. Windows, Linux, Sun Solaris, Mac/OS etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere(WORA).

Secured

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- No explicit pointer
- Java Programs run inside virtual machine sandbox



- **Classloader**: Classloader in Java is a part of the Java Runtime Environment(JRE) which is used to dynamically load Java classes into the Java Virtual Machine. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier**: It checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager**: It determines what resources a class can access such as reading and writing to the local disk.

These security are provided by java language. Some security can also be provided by application developer through SSL, JAAS, Cryptography etc.

Robust

- Robust simply means strong. Java is robust because:
- It uses strong memory management.
- There are lack of pointers that avoids security problems.
- There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There is exception handling and type checking mechanism in java. All these points makes java robust.

Architecture-neutral

Java is architecture neutral because there is no implementation dependent features e.g. size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. But in java, it occupies 4 bytes of memory for both 32 and 64 bit architectures.

Portable

Java is portable because it facilitates you to carry the java bytecode to any platform. It doesn't require any type of implementation.

High-performance

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g. C++). Java is an interpreted language that is why it is slower than compiled languages e.g. C, C++ etc.

Distributed

Java is distributed because it facilitates users to create distributed applications in java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications etc.

Dynamic

Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages i.e. C and C++.

Java supports dynamic compilation and automatic memory management (garbage collection).

1.2. Garbage Collection

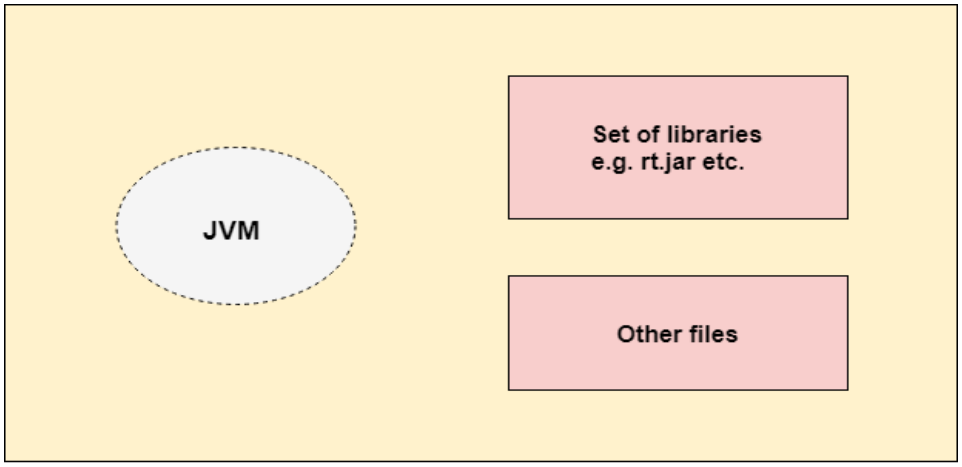
Objects are dynamically allocated by using the **new** operator, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation automatically this is called garbage collection. When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used.

1.3. The Java Environment

JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing java applications. It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime.

Implementation of JVMs are also actively released by other companies besides Sun Micro Systems.



JRE

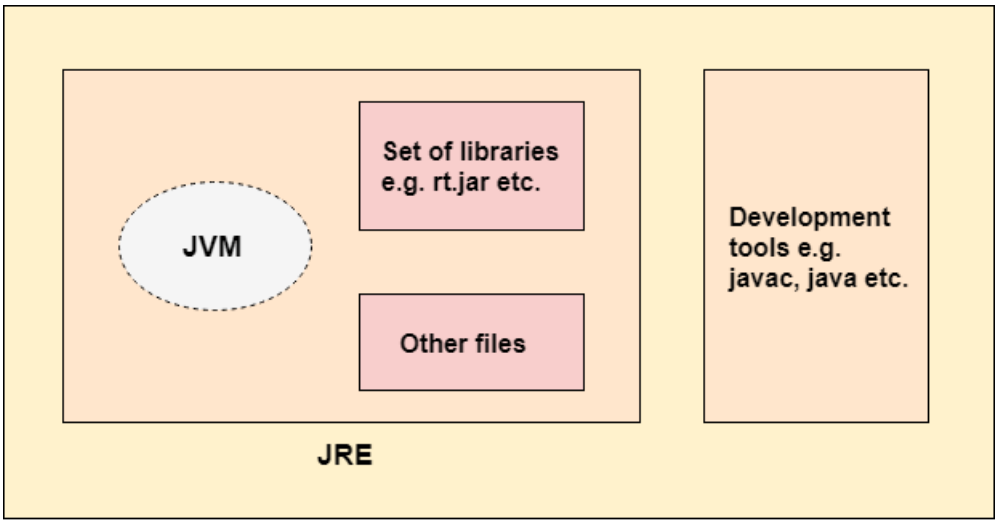
JDK

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

1.1 JVM (Java Virtual Machine)

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

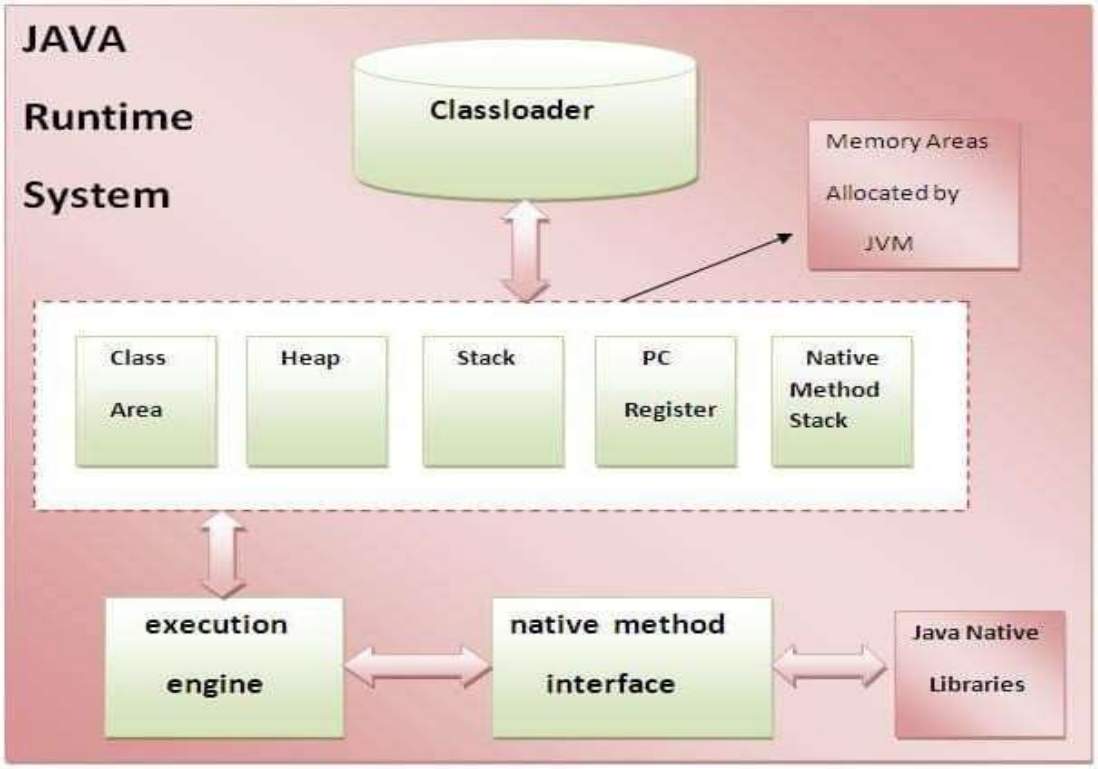
JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

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.



1.4. Internal Architecture Of Jvm

a) **Classloader**

Classloader is a subsystem of JVM that is used to load class files.

b) **Class(Method)** Area

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

c) Heap

It is the runtime data area in which objects are allocated.

d) Stack

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.

Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

e) Program Counter Register

PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

f) Native Method Stack

It contains all the native methods used in the application.

g) Execution Engine

Contains a virtual processor, Interpreter to read bytecode stream then execute the instructions and Just-In-Time(JIT) compiler is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

1.5. Structure Of Java Program

A first Simple Java Program

```
class Simple
{
   public static void main(String args[])
  {
    System.out.println("Java World");
   }
}
```

To compile:

javac Simple.java

To execute:

java Simple

class keyword is used to declare a class in java.

public keyword is an access modifier which represents visibility, it means it is visible to all.

static is a keyword, if we declare any method as static, it is known as static method. The core advantage of static method is that there is no need to create object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create object to invoke the main method. So it saves memory.

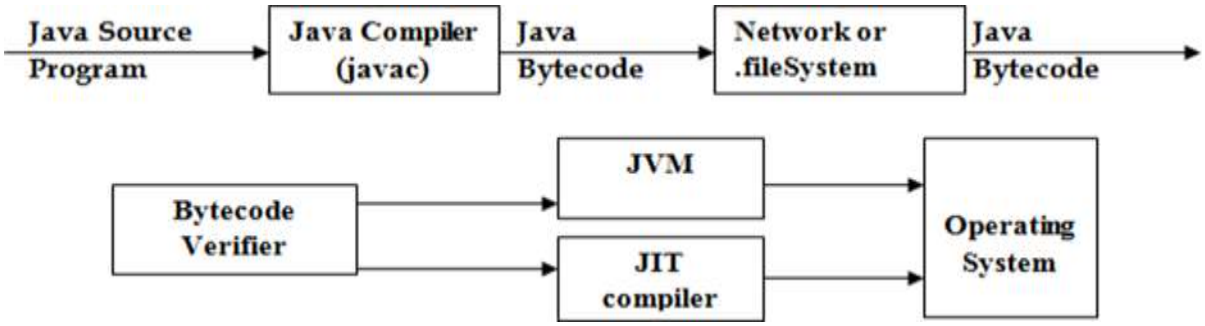
void is the return type of the method, it means it doesn't return any value.

main represents the starting point of the program. **String[] args** is used for command line argument. **System.out.println()** is used print statement.

A program is written in JAVA, the javac compiles it. The result of the JAVA compiler is the .class file or the bytecode and not the machine native code (unlike C compiler).

The bytecode generated is a non-executable code and needs an interpreter to execute on a machine. This interpreter is the JVM and thus the Bytecode is executed by the JVM.

And finally program runs to give the desired output.



CHAPTER TWO

2. BASICS IN JAVA PROGRAMMING

2.1. Java Comments

The java comments are statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code for specific time.

There are 3 types of comments in java.

1. Single Line Comment
2. Multi Line Comment
3. Documentation Comment

Java Single Line Comment

The single line comment is used to comment only one line.

Syntax:

```
//This is single line comment
```

Example:

```
public class CommentExample1
{
 public static void main(String[] args)
 {
   int i=10;//Here, i is a variable
   System.out.println(i);
 }
}
```

Output:

```
10
```

Java Multi Line Comment

The multi line comment is used to comment multiple lines of code.

Syntax:

```
/*
This
is
multi line
comment
*/
```

Example:

```
public class CommentExample2{
  public static void main(String[] args)
  {
    /* Let's declare and
     print variable in java. */
     int i=10;
     System.out.println(i);
  }
}
```

Output:

```
10
```

Java Documentation Comment

The documentation comment is used to create documentation API. To create documentation API, you need to use **javadoc tool.**

Syntax:

```
/**
This
is
documentation
comment
*/
```

Example:

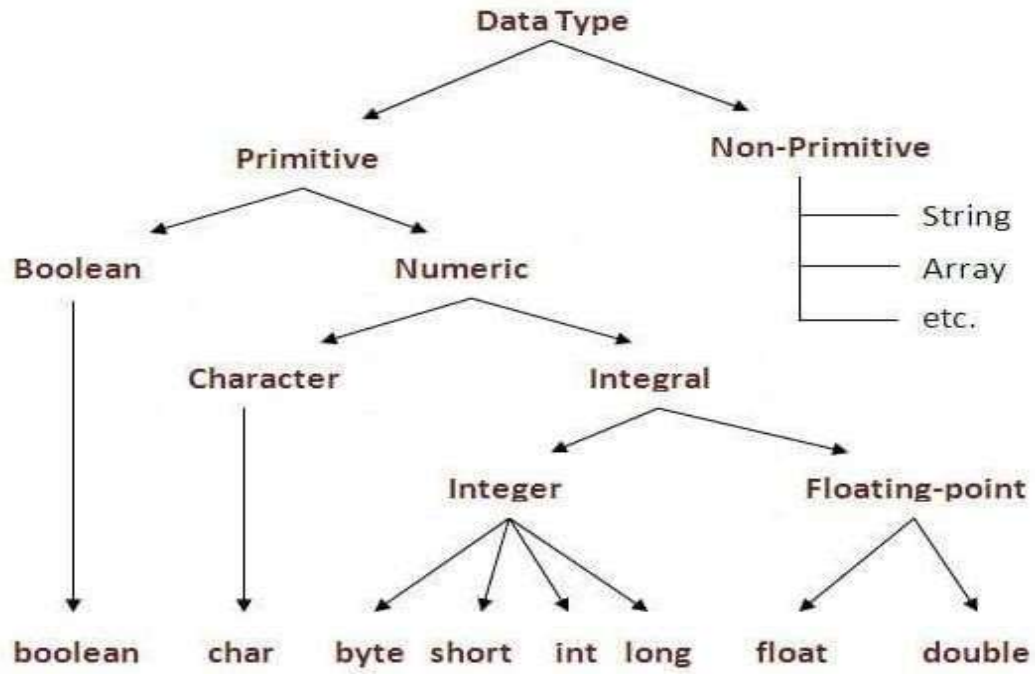
```
/** The Calculator class provides methods to get addition and subtraction of given 2 numbers.*/
public class Calculator
{
/** The add() method returns addition of given numbers.*/
        public static int add(int a, int b)
        {
        return a+b;
        }
/** The sub() method returns subtraction of given numbers.*/
        public static int sub(int a, int b)
        {
        return a-b;
        }
}
```

This type of comment is used to produce an HTML file that documents your program. The documentation comment begins with a /** and ends with a */.

2.1. **Datatypes In Java**

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include Integer, Character, Boolean, and Floating Point.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.



Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean. These can be put in four groups:

• Integers :- This group includes byte, short, int, and long, which are for whole-valued signed numbers.

• Floating-point numbers :- This group includes float and double, which represent numbers with fractionalprecision.

• Characters:- This group includes char, which represents symbols in a character set, like letters and numbers.

• Boolean :- This group includes boolean, which is a special type for representing true/false values.

Example :

```
// Compute distance light travels using long variables.
class Light
{
 public static void main(String args[])
 {
 int lightspeed;
 long days;
 long seconds;
 long distance;
 // approximate speed of light in miles per second
 lightspeed = 186000;
```

```
days = 1000; // specify number of days here
seconds = days * 24 * 60 * 60; // convert to seconds
distance = lightspeed * seconds; // compute distance
System.out.print("In " + days);
System.out.print(" days light will travel about ");
System.out.println(distance + " miles.");
}
}
```

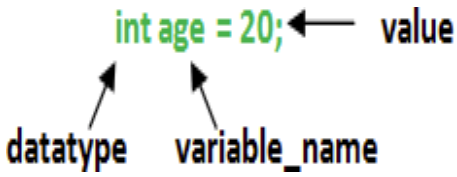
Output:

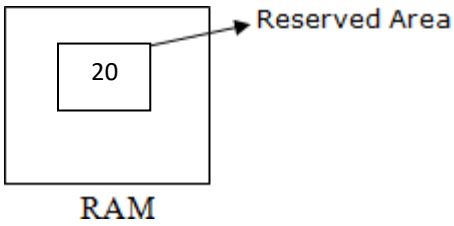
In 1000 days light will travel about 16070400000000 miles.

Clearly, the result could not have been held in an int variable.

2.2.**Variables**

A variable is a container which holds the value and that can be changed durig the execution of the program. A variable is assigned with a datatype. Variable is a name of memory location. All the variables must be declared before they can be used. There are three types of variables in java: local variable, instance variable and static variable.





a) Local Variable

A variable defined within a block or method or constructor is called local variable.

- These variable are created when the block in entered or the function is called and destroyed after exiting from the block or when the call returns from the function.
- The scope of these variables exists only within the block in which the variable is declared. i.e. we can access these variable only within that block.

Example:

```
import java.io.*;
public class StudentDetails
{
   public void StudentAge()
   { //local variable age
      int age = 0;
      age = age + 5;
      System.out.println("Student age is : " + age);
   }
   public static void main(String args[])
   {
      StudentDetails obj = new StudentDetails();
      obj.StudentAge();
   }
}
```

Output:

Student age is : 5

b) Instance Variable

Instance variables are non-static variables and are declared in a class outside any method, constructor or block.

a. As instance variables are declared in a class, these variables are created when an object of the classis created and destroyed when the object is destroyed.

b. Unlike local variables, we may use access specifiers for instance variables. If we do not specify anyaccess specifier then the default access specifier will be used.

Example:

```
import java.io.*;
class Marks
{
   int m1;
   int m2;
}
class MarksDemo
{
   public static void main(String args[])
   {  //first object
      Marks obj1 = new Marks();
      obj1.m1 = 50;
      obj1.m2 = 80;
      //second object
      Marks obj2 = new Marks();
      obj2.m1 = 80;
      obj2.m2 = 60;
      //displaying marks for first object
      System.out.println("Marks for first object:");
      System.out.println(obj1.m1);
      System.out.println(obj1.m2);
      //displaying marks for second object
      System.out.println("Marks for second object:");
      System.out.println(obj2.m1);
      System.out.println(obj2.m2);
   }}
```

Output:

```
Marks for first object:
50
80
Marks for second object:
80
60
```

c) Static variable

Static variables are also known as Class variables.

a. These variables are declared similarly as instance variables, the difference is that static variables aredeclared using the static keyword within a class outside any method constructor or block.
b. Unlike instance variables, we can only have one copy of a static variable per class irrespective ofhow many objects we create.
c. Static variables are created at start of program execution and destroyed automatically when execution ends.

Example:

```
import java.io.*;
class Emp {

  // static variable salary
  public static double salary;
  public static String name = "Vijaya";
}
public class EmpDemo
{
    public static void main(String args[]) {

     //accessing static variable without object
     Emp.salary = 1000;
     System.out.println(Emp.name + "'s average salary:" + Emp.salary);
  }
 }
```

Output:

Vijaya’s average salary:10000.0

Difference between Instance variable and Static variable

INSTANCE VARIABLE	STATIC VARIABLE
Each object will have its **own copy** of instance variable	We can only have **one copy** of a static variable per class irrespective of how many objects we create.
Changes made in an instance variable using one object will **not be reflected** in other objects as each object has its own copy of instance variable	In case of static changes **will be reflected** in other objects as static variables are common to all object of a class.
We can access instance variables **through object references**	Static Variables can be accessed **directly using class name.**

Class Sample
{
int a;
}
	Class Sample
	{
	static int a;
	}

2.3.**Operators In Java**

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups –

i. Arithmetic Operators
ii. Increment and Decrement
iii. Bitwise Operators
iv. Relational Operators
v. Boolean Operators
vi. Assignment Operator
vii. Ternary Operator

2.3.1. **Arithmetic Operators**

Arithmetic operators are used to manipulate mathematical expressions.

Operator	Result
+	Addition (also unary plus)
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
––	Decrement

Example:

```
// Demonstrate the basic arithmetic operators.
class BasicMath
{
public static void main(String args[])
{
// arithmetic using integers
System.out.println("Integer Arithmetic");
int a = 1 + 1;
int b = a * 3;
int c = b / 4;
int d = c - a;
int e = -d;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
System.out.println("e = " + e);
// arithmetic using doubles
System.out.println("\nFloating Point Arithmetic");
double da = 1 + 1;
```

```
double db = da * 3;
double dc = db / 4;
double dd = dc - a;
double de = -dd;
System.out.println("da = " + da);
System.out.println("db = " + db);
System.out.println("dc = " + dc);
System.out.println("dd = " + dd);
System.out.println("de = " + de);
}}
```

Output:

```
Integer Arithmetic
a = 2
b = 6
c = 1
d = -1
e = 1
Floating Point Arithmetic
da = 2.0
db = 6
dc = 1.5
dd = -0.5
de = 0.5
```

2.3.2. **Modulus Operator**

The modulus operator, **%**, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types.

Example:

```
// Demonstrate the % operator.
class Modulus {
public static void main(String args[])
{
int x = 42;
double y = 42.25;
System.out.println("x mod 10 = " + x % 10);
System.out.println("y mod 10 = " + y % 10);
}
```

```
}
```

Output:

```
x mod 10 = 2
y mod 10 = 2.25
```

2.3.3. Arithmetic Compound Assignment Operators

Java provides special operators that can be used to combine an arithmetic operation with an assignment.

```
a = a + 4;
```

In Java, you can rewrite this statement as shown here:

```
a += 4;
```

Syntax:

var op= expression;

Example:

```
// Demonstrate several assignment operators.
class OpEquals
{
public static void main(String args[])
{
int a = 1;
int b = 2;
int c = 3;
a += 5;
b *= 4;
c += a * b;
c %= 6;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
}
}
```

Output:

a = 6
b = 8
c = 3

2.3.4. Increment and Decrement Operators

The ++ and the – – are Java's increment and decrement operators. The increment operator increases its operand by one. The decrement operator decreases its operand by one.

Example:

```
// Demonstrate ++.
class IncDec
{
public static void main(String args[])
{
int a = 1;
int b = 2;
int c;
int d;
c = ++b;
d = a++;
c++;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
}
}
```

Output:

a = 2
b = 3
c = 4
d = 1

2.3.5. **Bitwise Operators**

Java defines several *bitwise operators* that can be applied to the integer types: **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands.

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

*2.3.5.1.**Bitwise Logical Operators***

The bitwise logical operators are **&**, |, ^, and **~**. the bitwise operators are applied to each individual bit within each operand.

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

Example:

```
// Demonstrate the bitwise logical operators.
class BitLogic
{
public static void main(String args[])
{
String binary[] = {"0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
"1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"};
int a = 3; // 0 + 2 + 1 or 0011 in binary
int b = 6; // 4 + 2 + 0 or 0110 in binary
```

```
int c = a | b;
int d = a & b;
int e = a ^ b;
int f = (~a & b)|(a & ~b);
int g = ~a & 0x0f;
System.out.println(" a = " + binary[a]);
System.out.println(" b = " + binary[b]);
System.out.println(" a|b = " + binary[c]);
System.out.println(" a&b = " + binary[d]);
System.out.println(" a^b = " + binary[e]);
System.out.println("~a&b|a&~b = " + binary[f]);
System.out.println(" ~a = " + binary[g]);
}
}
```

Output:

```
a = 0011
b = 0110
a|b = 0111
a&b = 0010
a^b = 0101
~a&b|a&~b = 0101
~a = 1100
```

Left Shift Operator

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

Example:

```
class OperatorExample
{
public static void main(String args[])
{
System.out.println(10<<2);//10*2^2=10*4=40
System.out.println(10<<3);//10*2^3=10*8=80
System.out.println(20<<2);//20*2^2=20*4=80
System.out.println(15<<4);//15*2^4=15*16=240
}
```

Output:

40

80

80

240

Right Shift Operator

The Java right shift operator >> is used to move left operands value to right by the number of bits specified by the right operand.

Example:

```
class OperatorExample
{
public static void main(String args[])
{
System.out.println(10>>2);//10/2^2=10/4=2
System.out.println(20>>2);//20/2^2=20/4=5
System.out.println(20>>3);//20/2^3=20/8=2
}
}
```

Output:

2

5

2

2.3.6. **Relational Operators**

The relational operators determine the relationship that one operand has to the other.Specifically, they determine equality and ordering. The outcome of these operations is a **boolean** value.

2.3.7. **Boolean Operators**

The Boolean logical operators shown here operate only on boolean operands. All of the binary logical operators combine two boolean values to form a resultant boolean value.Example:

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

```
// Demonstrate the boolean logical operators.
class BoolLogic
{
public static void main(String args[])
{
boolean a = true;
boolean b = false;
boolean c = a | b;
boolean d= a & b;
boolean e = a ^ b;
boolean f = (!a & b) | (a & !b);
boolean g = !a;
System.out.println(" a = " + a);
System.out.println(" b = " + b);
System.out.println(" a|b = " + c);
System.out.println(" a&b = " + d);
System.out.println(" a^b = " + e);
System.out.println("!a&b|a&!b = " + f);
System.out.println(" !a = " + g);
}
```

```
}
```

Output:

```
a =  true
b = false
a|b = true
a&b = false
a^b = true
!a&b|a&!b = true
!a=false
```

In the output, the string representation of a Java **boolean** value is one of the literal values **true** or **false**. Java AND Operator Example: Logical && and Bitwise &

The logical && operator doesn't check second condition if first condition is false. It checks second condition only if first one is true.

The bitwise & operator always checks both conditions whether first condition is true or false.

Example:

```
class OperatorExample
{
public static void main(String args[])
{
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a<c);//false && true = false
System.out.println(a<b&a<c);//false & true = false
}
}
```

Output:

```
false
false
```

2.3.8. **Assignment Operator**

The assignment operator is the single equal sign, =.

Syntax:

var = expression*;*

Here, the type of *var* must be compatible with the type of *expression*.

int x, y, z;

x = y = z = 100; // set x, y, and z to 100

This fragment sets the variables **x**, **y**, and **z** to 100 using a single statement.

2.3.9. **Ternary Operator**

Ternary operator in java is used as one liner replacement for if-then-else statement and used a lot in java programming. it is the only conditional operator which takes three operands.

Syntax:

expression1 ? expression2 : expression3

Example:

```
class OperatorExample
{
public static void main(String args[])
{
int a=2;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}
}
```

Output:

2

CHAPTER THREE

3. DECISION AND REPETITION STATEMENTS

3.1. Selection Statements In Java

A programming language uses control statements to control the flow of execution of program based on certain conditions.

3.2. Java's Selection statements:

- if
- if-else
- nested-if
- if-else-if
- switch-case
- jump – break, continue, return

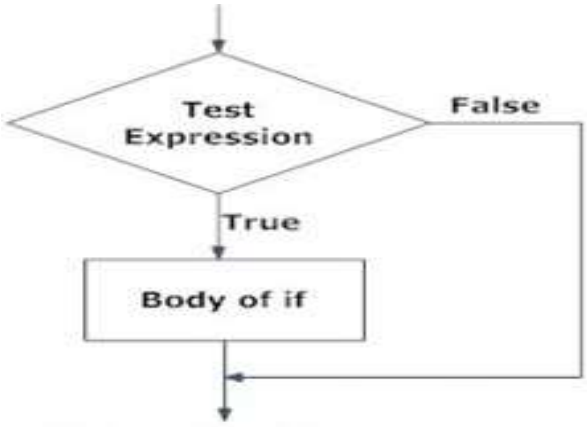
3.2.1. if Statement

if statement is the most simple decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not that is if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

```
if(condition)
{
    //statements to execute if
    //condition is true
}
```

Condition after evaluation will be either true or false. If the value is true then it will execute the block of statements under it. If there are no curly braces '{' and '}' after **if(condition)** then by default if statement will consider the immediate one statement to be inside its block.



Example:

```
class IfSample
{
public static void main(String args[])
{
int x, y;
x = 10;
y = 20;
if(x < y)
System.out.println("x is less than y");
x = x * 2;
if(x == y)
System.out.println("x now equal to y");
x = x * 2;
if(x > y)
System.out.println("x now greater than y");
// this won't display anything
if(x == y)
System.out.println("you won't see this");
}
}
```

Output:

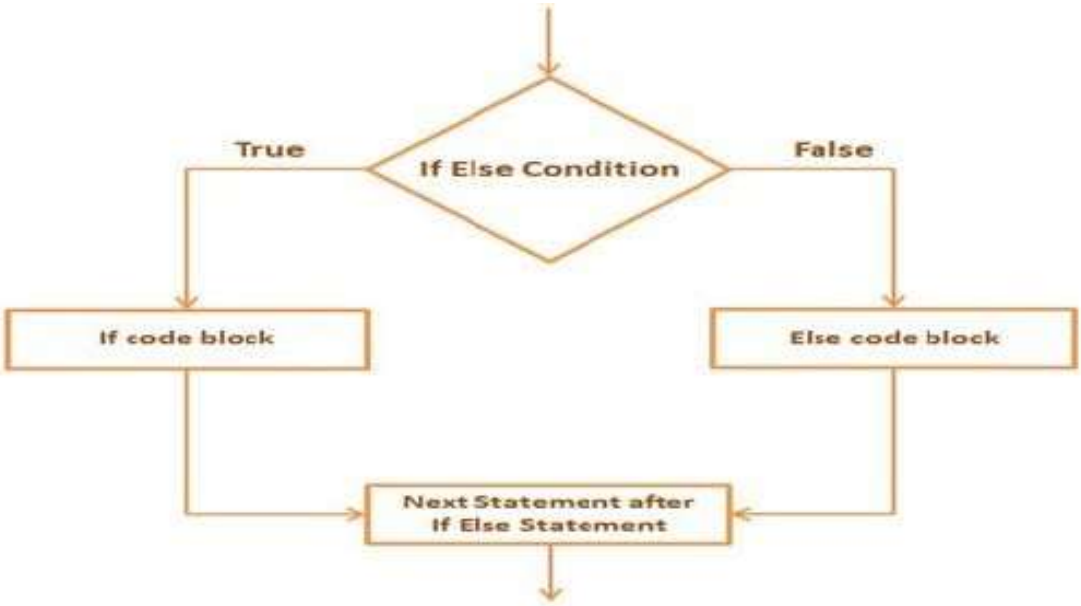
```
x is less than y
x now equal to y
x now greater than y
```

3.2.2. **if-else Statement**

The Java if-else statement also tests the condition. It executes the *if* block if condition is true else if it is false the else block is executed.

Syntax:

```
if(condition)
{
        //Executes this block if
        //condition is true
}
else
{
        //Executes this block if
        //condition is false

}
```



Example:

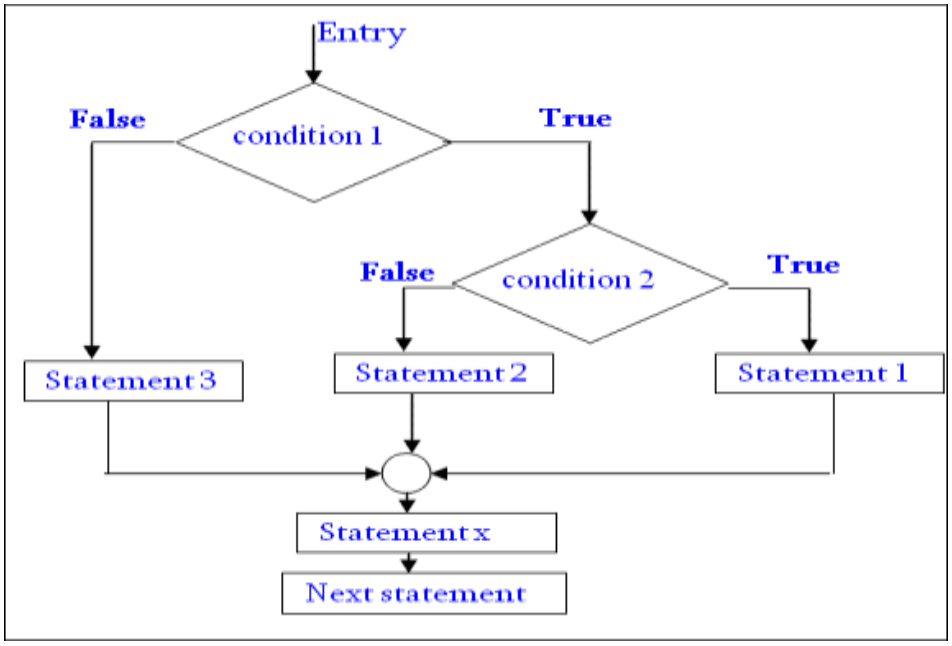
```
public class IfElseExample
{
   public static void main(String[] args)
  {
   int number=13;
   if(number%2==0)
   {
      System.out.println("even number");
   }else
  {
      System.out.println("odd number");
  }
  }
  }
```

Output:

odd number

3.2.3. **Nested if Statement**

Nested if-else statements, is that using one if or else if statement inside another if or else if statement(s).



Example:

```
// Java program to illustrate nested-if statement
class NestedIfDemo
{
   public static void main(String args[])
   {
      int i = 10;

      if (i == 10)
      {
         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");
      }
   }
}
```

Output:

i is smaller than 15

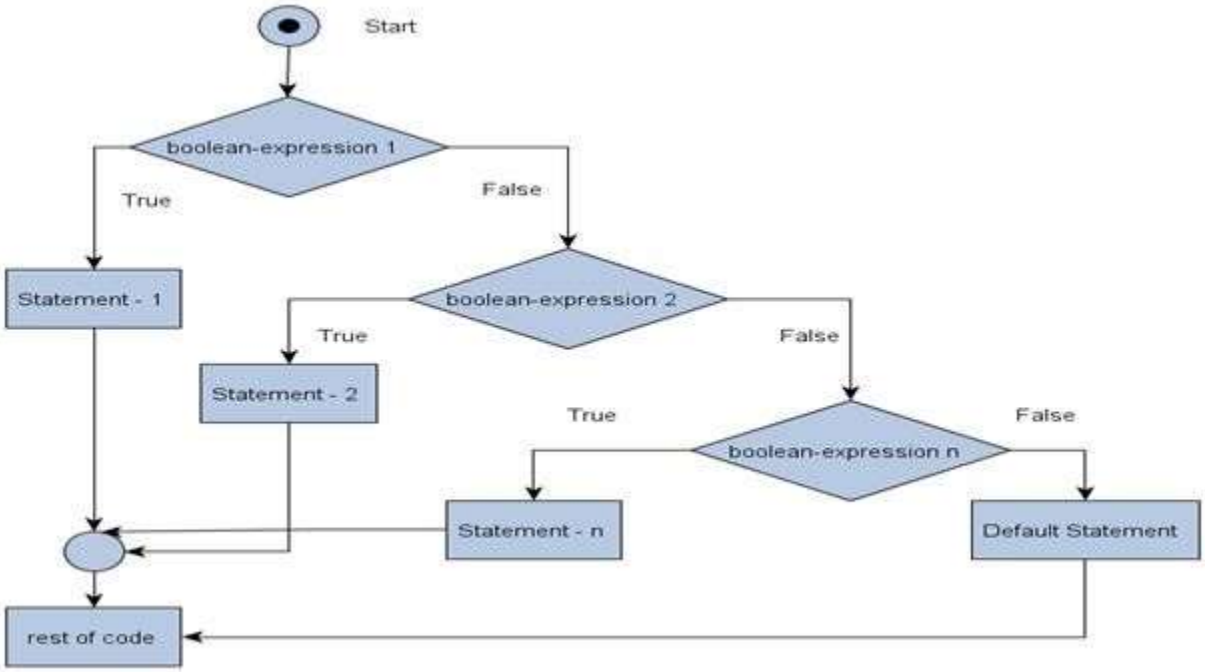
i is smaller than 12 too

3.2.4. **if-else-if ladder statement**

The *if* statements are executed from the top down. 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.

Syntax:

```
if(condition)
statement;
else if(condition)
statement;
else if(condition)
statement;
.
.
else
statement;
```



Example:

```
public class IfElseIfExample {
public static void main(String[] args) {
    int marks=65;
    if(marks<50){
        System.out.println("fail");
    }
    else if(marks>=50 && marks<60){
    System.out.println("D grade");
    }
    else if(marks>=60 && marks<70){
        System.out.println("C grade");
    }
    else if(marks>=70 && marks<80){
        System.out.println("B grade");
    }
    else if(marks>=80 && marks<90){
        System.out.println("A grade");
    }else if(marks>=90 && marks<100){
        System.out.println("A+ grade");
    }else{
        System.out.println("Invalid!");
    }
 }
}
```

Output:

C grade

3.2.5. **Switch Statements**

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

Syntax:

```
switch (expression) {
case value1:
// statement sequence
break;
case value2:
// statement sequence
break;
.
.
case valueN :
// statement sequence
break;
default:
// default statement sequence
}
```

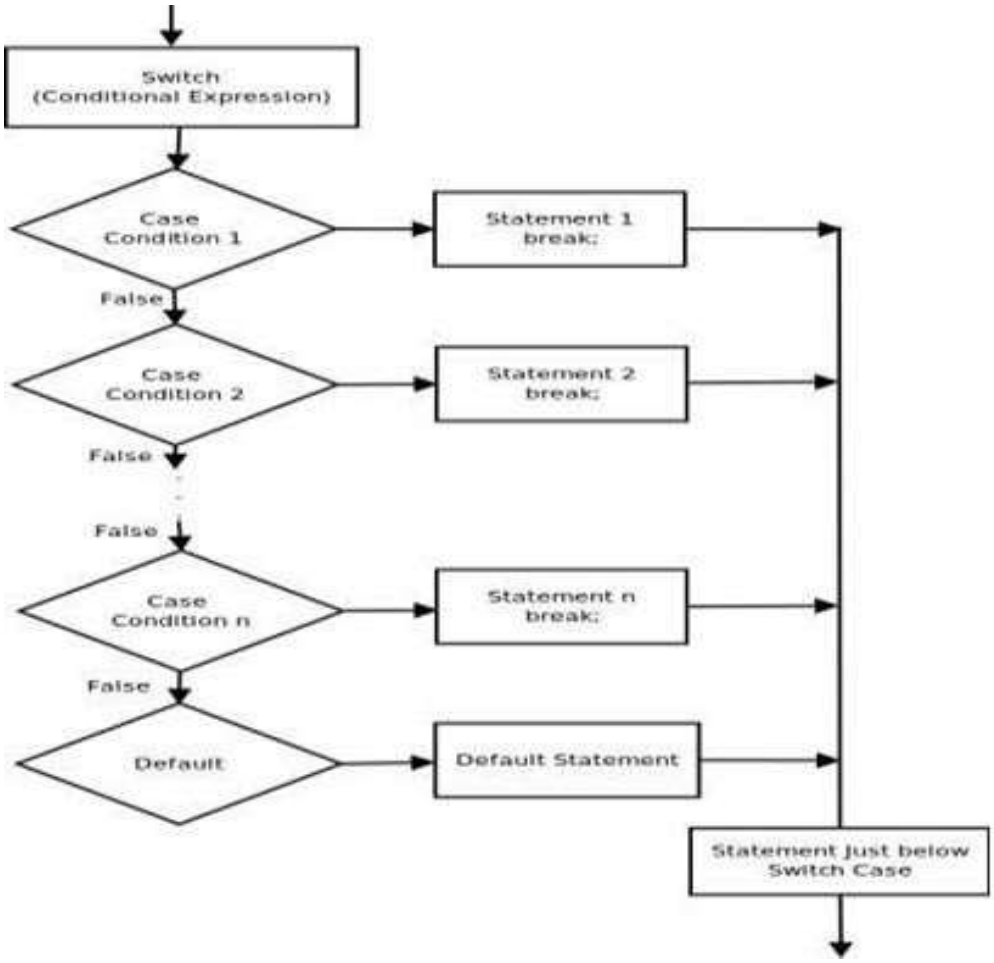
Example:

```
// A simple example of the switch.
class SampleSwitch
 {
public static void main(String args[])
 {
for(int i=0; i<6; i++)
switch(i) {
case 0:
System.out.println("i is zero.");
break;
```

```
case 1:
System.out.println("i is one.");
break;
case 2:
System.out.println("i is two.");
break;
case 3:
System.out.println("i is three.");
break;
default:
System.out.println("i is greater than 3.");
}
}
}
```

Output:

```
i is zero.
i  is  one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```



3.3.**Iterative Statements**

In programming languages, loops are used to execute a set of instructions/functions repeatedly when some conditions become true. There are three types of loops in java.

- while loop
- do-while loop
- For loop

3.3.1. **while loop**

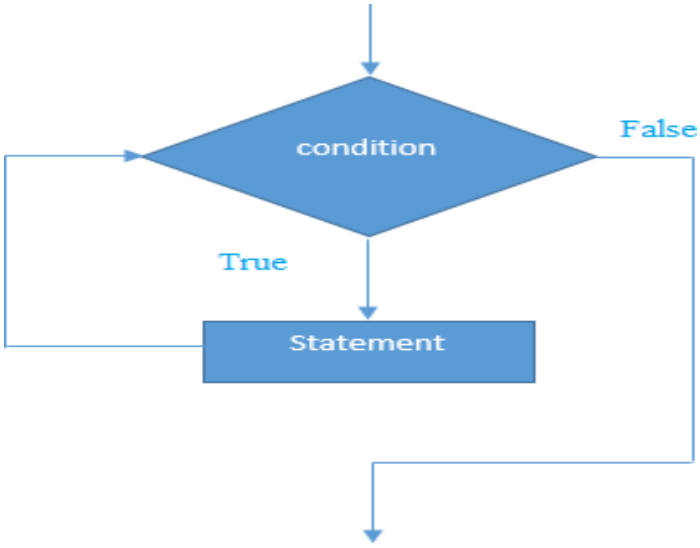
A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.

Syntax:

while(*condition*) {

// body of loop

}



- While loop starts with the checking of condition. If it evaluated to true, then the loop body statements are executed otherwise first statement following the loop is executed. It is called as Entry controlled loop.
- Normally the statements contain an update value for the variable being processed for the next iteration.
- When the condition becomes false, the loop terminates which marks the end of its life cycle.

Example:

```
// Demonstrate the while loop.
class While {
public static void main(String args[]) {
int n = 5;
while(n > 0) {
System.out.println("tick " + n);
n--;
}
}
}
```

Output:

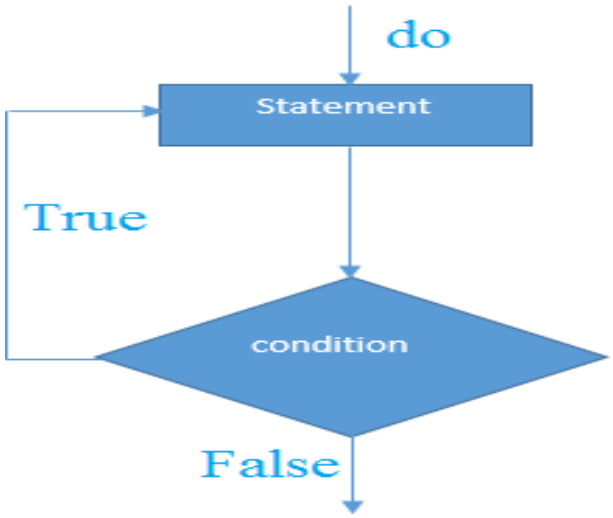
```
tick 5
tick 4
tick 3
tick 2
tick 1
```

3.3.2. **do-while loop:**

do while loop checks for condition after executing the statements, and therefore it is called as Exit Controlled Loop**.**

Syntax:

```
do {
// body of loop
} while (condition);
```



- 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 true, next iteration of 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 atleast once before any condition is checked, and therefore is an example of exit control loop.

Example

```
public class DoWhileExample {
public static void main(String[] args) {
   int i=1;
   do{
      System.out.println(i);
   i++;
   }while(i<=5);
}
}
```

Output:

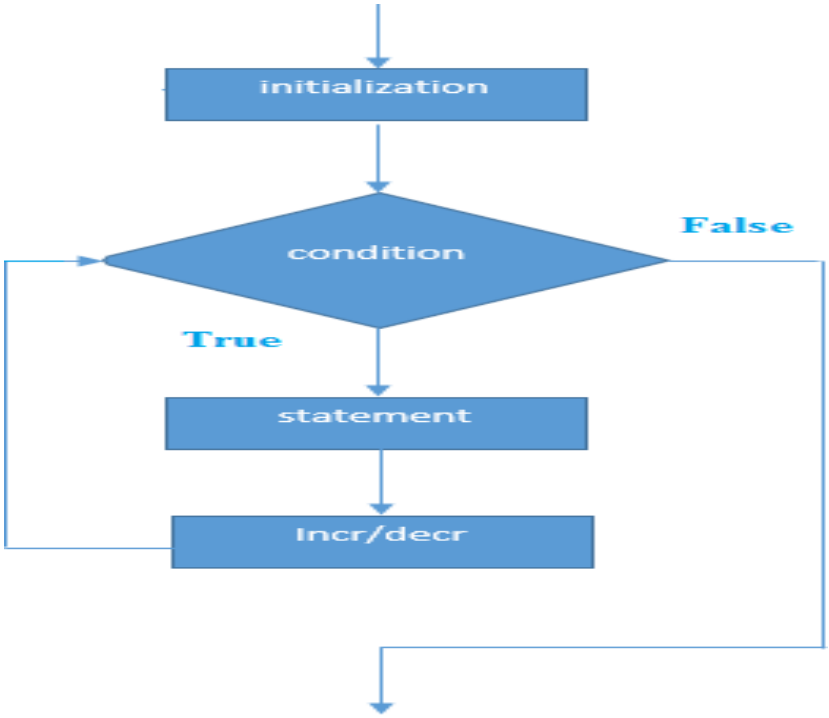
1
2
3
4
5

3.3.3. **for loop**

for loop provides a concise way of writing the loop structure. A for statement consumes the initialization, condition and increment/decrement in one line.

Syntax

```
for(initialization; condition; iteration) {
// body
}
```



- **Initialization condition:** Here, we initialize the variable in use. It marks the start of a for loop. An already declared variable can be used or a variable can be declared, local to loop only.
- **Testing Condition:** It is used for testing the exit condition for a loop. It must return a boolean value. It is also an **Entry Control Loop** as the condition is checked prior to the execution of the loop statements.
- **Statement execution:** Once the condition is evaluated to true, the statements in the loop body are executed.

- **Increment/ Decrement:** It is used for updating the variable for next iteration.
- **Loop termination:**When the condition becomes false, the loop terminates marking the end of its life cycle.

Example

```
public class ForExample
 {
   public static void main(String[] args)
{
   for(int i=1;i<=5;i++){
       System.out.println(i);
}
}
}
```

Output:

```
1
2
3
4
5
```

3.3.4. **for-each Loop**

The for-each loop is used to traverse array or collection in java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation. It works on elements basis not index.It returns element one by one in the defined variable.

Syntax:

```
for(type itr-var : collection) statement-block
```

Example:

```
// Use a for-each style for loop.
class ForEach {
public static void main(String args[]) {
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
// use for-each style for to display and sum the values
for(int x : nums) {
```

```
System.out.println("Value is: " + x);
sum += x;
}
System.out.println("Summation: " + sum);
}
}
```

Output:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Value is: 8
Value is: 9
Value is: 10
Summation: 55
```

3.3.5. **Nested Loops**

Java allows loops to be nested. That is, one loop may be inside another.

Example:

```
// Loops may be nested.
class Nested {
public static void main(String args[]) {
int i, j;
for(i=0; i<10; i++) {
for(j=i; j<10; j++)
System.out.print(".");
System.out.println();
}
}
}
```

Output:

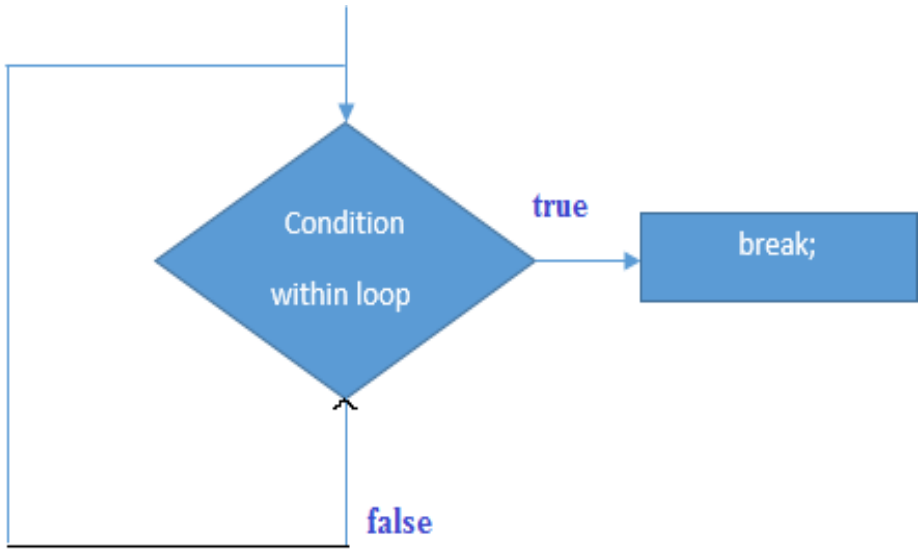
```
..........
.........
........
.......
......
.....
....
...
..
```

.

3.3.6. JUMP STATEMENTS

3.3.7. Java Break Statement

- When a break statement is encountered inside a loop, the loop is immediately terminated and theprogram control resumes at the next statement following the loop.
- The Java *break* is used to break loop or switch statement. It breaks the current flow of the program atspecified condition. In case of inner loop, it breaks only inner loop.



Example:

```
// Using
break to
exit a loop.
class
BreakLoop
{
public static void
main(String args[])
 {
```

```
for(int i=0; i<100;
i++)
{
if(i == 10) break; //
terminate loop if i is 10
System.out.println("i: " +
i);
}
System.out.println("Loop complete.");
}
}
```

Output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

3.3.8. **Java Continue Statement**

- The continue statement is used in loop control structure when you need to immediately jump to the next iteration of the loop. It can be used with for loop or while loop.
- The Java *continue statement* is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

Example:

```
// Demonstrate continue.class Continue
{
public static void main(String args[])
{
for(int i=0; i<10; i++)
{
 System.out.print(i + " ");
if (i%2 == 0)
 continue;System.out.println("");
}
}
}
```

This code uses the **%** operator to check if **i** is even. If it is, the loop continues withoutprinting a newline.

Output:

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

3.3.9. **Return**

The last control statement is return. The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

Example:

```
// Demonstrate return.class Return {
public static void main(String args[]) {
boolean t = true; System.out.println("Before the return.");
 if(t) return; // return to caller
System.out.println("This won't execute.");
}}
```

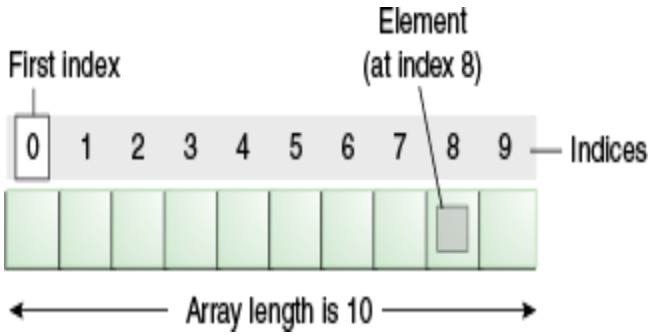
Output:

Before the return.

3.4.**Arrays**

- Array is a collection of similar type of elements that have contiguous memory location.

- ✓ In Java all arrays are dynamically allocated.
- ✓ Since arrays are objects in Java, we can find their length using member length.
- ✓ A Java array variable can also be declared like other variables with [] after the data type.
- ✓ The variables in the array are ordered and each have an index beginning from 0.
- ✓ Java array can be also be used as a static field, a local variable or a method parameter.
- ✓ The **size** of an array must be specified by an int value and not long or short.
- ✓ The direct superclass of an array type is Object.
- ✓ Every array type implements the interfaces Cloneable and java.io.Serializable.



3.4.1. **Advantage of Java Array**

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
- **Random access:** We can get any data located at any index position.

3.4.2. **Disadvantage of Java Array**

- **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

3.5.**Types of Array in java**

1. One- Dimensional Array
2. Multidimensional Array

3.5.1. **One-Dimensional Arrays**

An array is a group of like-typed variables that are referred to by a common name. An array declaration has two components: the type and the name. *type* declares the element type of the array. The element type determines the data type of each element that comprises the array. We can also create an array of other primitive data types like char, float, double..etc or user defined data type(objects of a class).Thus, the element

type for the array determines what type of data the array will hold.

Syntax:

type var-name[ ];

Instantiation of an Array in java

array-var = new type [size];

Example:

```
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}
}
```

Output:

10

20

70

40

50

Declaration, Instantiation and Initialization of Java ArrayExample:

```
class Testarray1
{
public static void main(String args[])
{
int a[]={33,3,4,5};//declaration, instantiation and initialization
```

```
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}
}
```

Output:

```
33
3
4
5
```

Passing Array to method in java

We can pass the java array to method so that we can reuse the same logic on any array.

Example:

```
class Testarray2{
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
 if(min>arr[i])
  min=arr[i];
 System.out.println(min);
}
public static void main(String args[]){
int a[]={33,3,4,5};
min(a);//passing array to method
}}
```

Output:

```
3
```

3.5.2. Multidimensional Arrays

Multidimensional arrays are arrays of arrays with each element of the array holding the reference of other array. These are also known as Jagged Arrays. A multidimensional array is created by appending one set of

square brackets ([]) per dimension.

Syntax:

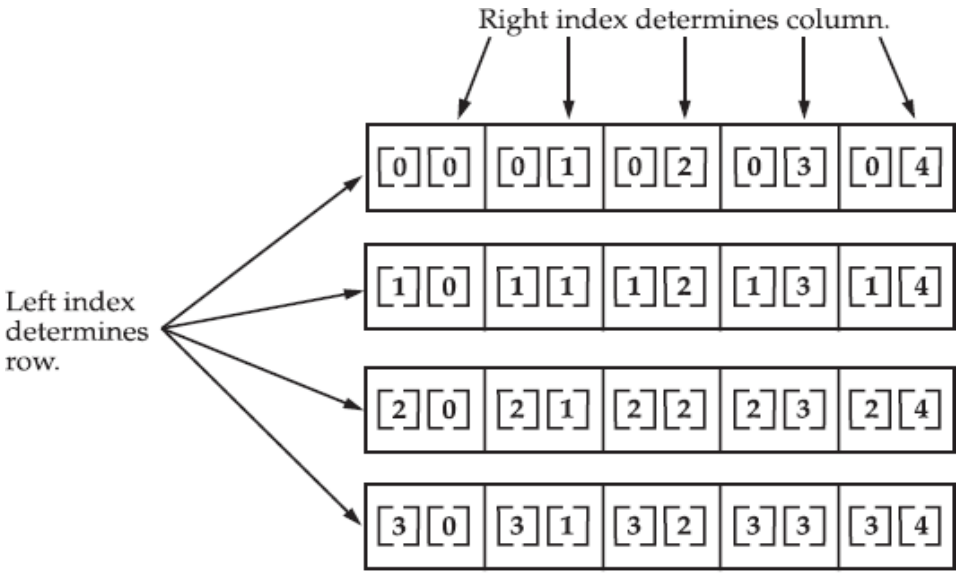
type var-name[ ][ ]=new type[row-size][col-size];

Example:

```
// Demonstrate a two-dimensional array.
class TwoDArray {
public static void main(String args[]) {
int twoD[][]= new int[4][5];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<5; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<5; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}
```

Output:

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```



Given: int twoD [ ] [ ] = new int [4] [5] ;

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of **twoD** when it is declared. It allocates the second dimension manually.

Syntax:

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

Example:

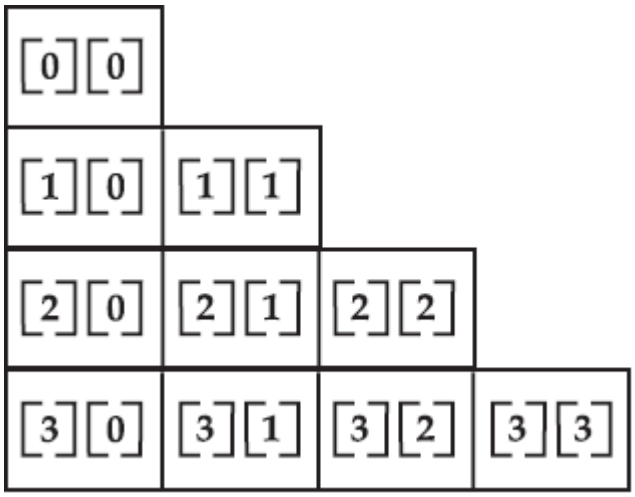
```
// Manually allocate differing size second dimensions.class TwoDAgain {
public static void main(String args[]) {int twoD[][] = new int[4][];
twoD[0] = new int[1];twoD[1] = new int[2];twoD[2] = new int[3];twoD[3] = new int[4];int i, j, k = 0;
for(i=0; i<4; i++) for(j=0; j<i+1; j++) {twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {for(j=0; j<i+1; j++)
System.out.print(twoD[i][j] + " ");System.out.println();
```

```
}

}

}
```

Output:

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

The array created by this program looks like this:



CHAPTER FOUR

4. OBJECTS AND CLASSES

4.1. Defining Classes In Java

The class is at the core of Java .A class is a *template* for an object, and an object is an *instance* of a class. A class is declared by use of the **class** keyword.

Syntax:

class *classname* {

type instance-variable1;
type instance-variable2;

// ...

type instance-variableN;

type methodname1(*parameter-list*) {

// body of method
}
...
type methodnameN(*parameter-list*) {

// body of method
}

The data, or variables, defined within a **class** are called *instance variables*. The code is contained within *methods*. The methods and variables defined within a class are called *members* of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class.

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another.

A Simple Class

class called **Box** that defines three instance variables: **width**, **height**, and **depth**.

class Box {
double width;
double height;
double depth;
}

The new data type is called **Box**. This name is used to declare objects of type **Box**. The class declaration only creates a template. It does not create an actual object.

To create a Box object

Box mybox = new Box(); // create a Box object called mybox

mybox will be an instance of **Box**.

Each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. To access these variables, you will use the *dot* (.) operator. The dot operator links the name of the object with the name of an instance variable.

Example1:

```
/* A program that uses the Box class.
Call this file BoxDemo.java
*/
class Box {
double width;
double height;
double depth;
}
// This class declares an object of type Box.
class BoxDemo {
public static void main(String args[]) {
Box mybox = new Box();
double vol;
// assign values to mybox's instance variables
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;
// compute volume of box
vol = mybox.width * mybox.height * mybox.depth;
System.out.println("Volume is " + vol);
}
}
```

Output:

Volume is 3000.0

Example2:

```
// This program declares two Box objects.
class Box {
double  width;
double height;
double depth;
}
class BoxDemo2 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// compute volume of first box
vol = mybox1.width * mybox1.height * mybox1.depth;
System.out.println("Volume is " + vol);
// compute volume of second box
vol = mybox2.width * mybox2.height * mybox2.depth;
System.out.println("Volume is " + vol);
}
}
```

Output:

```
Volume is 3000.0
Volume is 162.0
```

4.2.**Declaring Objects**

First, declare a variable of the class type. This variable does not define an object.Instead, it is simply a variable that can *refer* to an object.

Second, you must acquire an actual, physical copy of the object and assign it to that variable. This is done using the **new** operator. The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

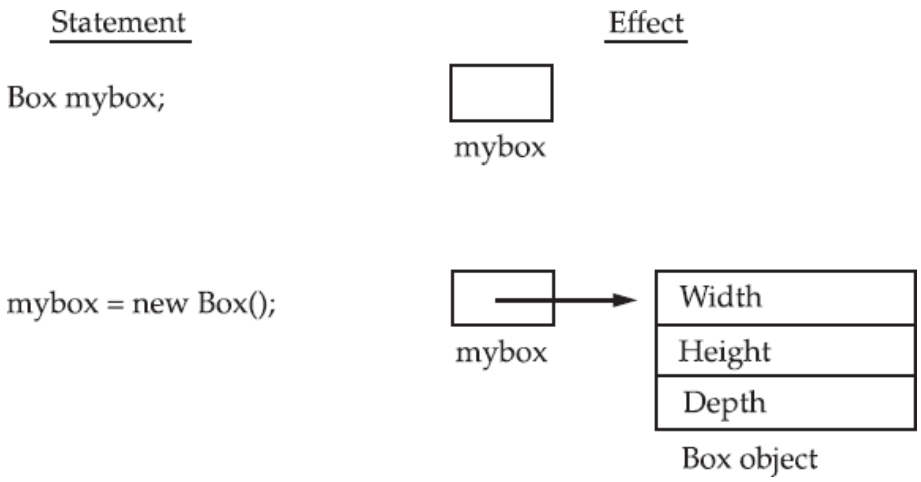
Syntax:

Box mybox = new Box();

Box mybox; // declare reference to object

mybox = new Box(); // allocate a Box object

The first line declares **mybox** as a reference to an object of type **Box**. At this point, **mybox** does not yet refer to an actual object. The next line allocates an object and assigns a reference to it to **mybox**. After the second line executes, we can use **mybox** as if it were a **Box** object. But in reality, **mybox** simply holds, in essence, the memory address of the actual **Box** object.



4.3. Assigning Object Reference

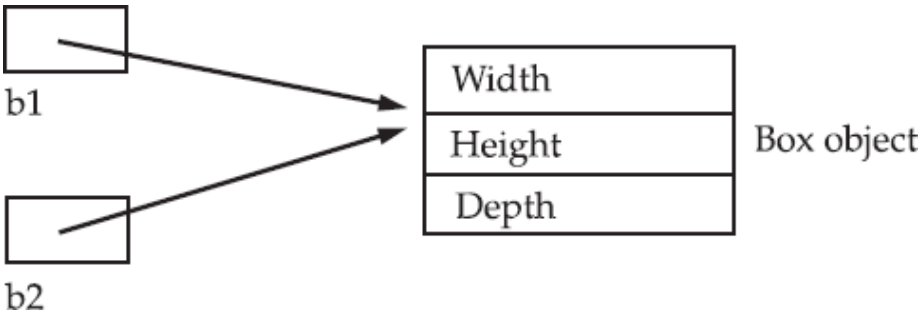
Variables Syntax:

Box b1 = new Box();

Box b2 = b1;

b2 is being assigned a reference to a copy of the object referred to by **b1**. **b1** and **b2** will both refer to the *same* object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original

object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.



4.4.**Constructors**

- Constructors are special member functions whose task is to initialize the objects of its class.
- It is a special member function, it has the same as the class name.
- Java constructors are invoked when their objects are created. It is named such because, it constructs the value, that is provides data for the object and are used to initialize objects.
- Every class has a constructor when we don't explicitly declare a constructor for any java class the compiler creates a default constructor for that class which does not have any return type.
- The constructor in Java cannot be abstract, static, final or synchronized and these modifiers are not allowed for the constructor.

There are two types of constructors:

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

4.4.1. **Default constructor (no-arg constructor)**

A constructor having no parameter is known as default constructor and no-arg constructor.

Example:

```
/* Here, Box uses a constructor to initialize the
dimensions of a box.
*/
class Box {
double width;
double height;
double depth;
```

```
// This is the constructor for Box.
Box() {
System.out.println("Constructing Box");
width = 10;
height = 10;
depth = 10;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo6 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

Output:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

new Box( ) is calling the **Box( )** constructor. When the constructor for a class is not explicitly defined , then Java creates a default constructor for the class. The default constructor automatically initializes all instance

variables to their default values, which are zero, **null**, and **false**, for numeric types, reference types, and **boolean**, respectively.

4.4.2. **Parameterized Constructors**

A constructor which has a specific number of parameters is called parameterized constructor. Parameterized constructor is used to provide different values to the distinct objects.

Example:

```
/* Here, Box uses a parameterized constructor to
initialize the dimensions of a box.
*/
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo7 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);
double vol;
// get volume of first box
vol = mybox1.volume();
```

```
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

Output:

```
Volume is 3000.0
Volume is 162.0
```

Box mybox1 = new Box(10, 20, 15);

The values 10, 20, and 15 are passed to the **Box( )** constructor when **new** creates the object. Thus, **mybox1**'s copy of **width**, **height**, and **depth** will contain the values 10, 20, and 15 respectively.

Overloading ConstructorsExample:

```
/* Here, Box defines three constructors to initialize
the dimensions of a box various ways.
*/
class Box {
double width;
double height;
double depth;
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
```

```
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class OverloadCons
{
public static void main(String args[])
{
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
}
}
```

Output:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

4.5.**Methods**

Syntax:

type name(parameter-list) {

// body of method

}

- *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create.
- If the method does not return a value, its return type must be **void**.
- The name of the method is specified by *name*.
- The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.
- Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

Syntax:

return *value*;

Example:

```
// This program includes a method inside the box class.
class Box {
double  width;
double height;
double depth;
// display volume of a box
void volume() {
System.out.print("Volume is ");
System.out.println(width * height * depth);
}
}
class BoxDemo3 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
```

```
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// display volume of first box
mybox1.volume();
// display volume of second box
mybox2.volume();
}
}
```

Output:

```
Volume is 3000.0
Volume is 162.0
```

The first line here invokes the **volume( )** method on **mybox1**. That is, it calls **volume( )** relative to the **mybox1** object, using the object's name followed by the dot operator. Thus, the call to **mybox1.volume( )** displays the volume of the box defined by **mybox1**, and the call to **mybox2.volume( )** displays the volume of the box defined by **mybox2**. Each time **volume( )** is invoked, it displays the volume for the specified box.

Returning a ValueExample:

```
// Now, volume() returns the volume of a box.
class Box {
double  width;
double height;
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
}
```

```
class BoxDemo4 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

Output:

```
Volume is 3000
Volume is 162
```

when **volume( )** is called, it is put on the right side of an assignment statement. On the left is a variable, in this case **vol**, that will receive the value returned by **volume( )**.

Syntax:

```
vol = mybox1.volume();
```

executes, the value of **mybox1.volume( )** is 3,000 and this value then is stored in **vol**.

There are two important things to understand about returning values:

• The type of data returned by a method must be compatible with the return type specified by the method.

• The variable receiving the value returned by a method (such as **vol**, in this case) must also be compatible with the return type specified for the method.

Adding a Method That Takes ParametersExample:

```
// This program uses a parameterized method.
class Box {
double  width;
double height;
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
// sets dimensions of box
void setDim(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
}
class BoxDemo5 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// initialize each box
mybox1.setDim(10, 20, 15);
mybox2.setDim(3, 6, 9);
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
```

```
System.out.println("Volume is " + vol);
}
}
```

Output:

Volume is 3000

Volume is 162

4.5.1. The this Keyword

This keyword is used to refer to the object that invoked it. This can be used inside any method to refer to the *current* object. That is, this is always a reference to the object on which the method was invoked. this() can be used to invoke current class constructor.

Syntax:

```
Box(double w, double h, double d) {
this.width = w;
this.height = h;
this.depth = d;
}
```

Example:

```
class Student
{
  int id;
  String name;
  student(int id, String name)
 {
    this.id = id;
    this.name = name;

}
void display()
{
        System.out.println(id+" "+name);
 }
 public static void main(String args[])
```

```
{
  Student stud1 = new Student(01,"Tarun");
  Student stud2 = new Student(02,"Barun");
  stud1.display();
  stud2.display();
 }
}
```

Output:

```
01 Tarun
02 Barun
```

4.6.**Access Protection**

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public

4.6.1. **Private Access Modifier**

The private access modifier is accessible only within class.

Simple example of private access modifier

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

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

```
  }
}
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

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

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

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

4.6.2. **Default Access Modifier**

If you don't use any modifier, it is treated as default by default. The default modifier is accessible only within package.

Example:

In this example, we have created two packages pack and mypack. We are accessing the A class from outside

its package, since A class is not public, so it cannot be accessed from outside the package.

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

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

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

4.6.3. **Protected Access Modifier**

The protected access modifier is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

Example:

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

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

```
//save by B.java
package mypack;
import pack.*;

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

Output:

Hello

4.6.4. **Public Access Modifier**

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

Example:

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

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

Output:

Hello

Access Modifier	Within Class	Within Package	Outside Package	Outside Package
			By Subclass Only	
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Java access modifiers with method overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

```
class A{
protected void msg(){System.out.println("Hello java");}
}
public class Simple extends A{
void msg(){System.out.println("Hello java");}//C.T.Error
 public static void main(String args[]){
   Simple obj=new Simple();
   obj.msg();
   }
}
```

The default modifier is more restrictive than protected. That is why there is compile time error.

4.7.**Static Members**

Static is a non-access modifier in Java which is applicable for the following:

1. blocks
2. variables
3. methods
4. nested classes

4.7.1. **Static blocks**

If you need to do computation in order to initialize your **static variables**, you can declare a static block that

gets executed exactly once, when the class is first loaded.

Example:

```
// Java program to demonstrate use of static blocks
class Test
{
   // static variable
   static int a = 10;
   static int b;

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

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

Output:

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

*4.7.2. **Static variables***

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

Important points for static variables :-

- We can create static variables at class-level only.

- static block and static variables are executed in order they are present in a program.

Example:

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
static int a = 3;
static int b;
static void meth(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {
System.out.println("Static block initialized.");
b = a * 4;
}
public static void main(String args[]) {
meth(42);
}
}
```

Output:

```
Static block initialized.
x = 42
a = 3
b = 12
```

4.7.3. **Static methods**

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

- They can only directly call other static methods.

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

Syntax:

classname.method( )

Example:

```
//Inside main( ), the static method callme( ) and the static variable b are accessed through their class name
//StaticDemo.
class StaticDemo {
static int a = 42;
static int b = 99;
static void callme() {
System.out.println("a = " + a);
}
}
class StaticByName {
public static void main(String args[]) {
StaticDemo.callme();
System.out.println("b = " + StaticDemo.b);
}
}
```

Output:

a = 42

b = 99

4.8.**Packages**

A java package is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form, built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.

Defining a Package

To create a package include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a namespace in which classes are stored. If **package** statement is omitted, the class names are put into the default package, which has no name.

Syntax:

package <fully qualified package name>;

package *pkg*;

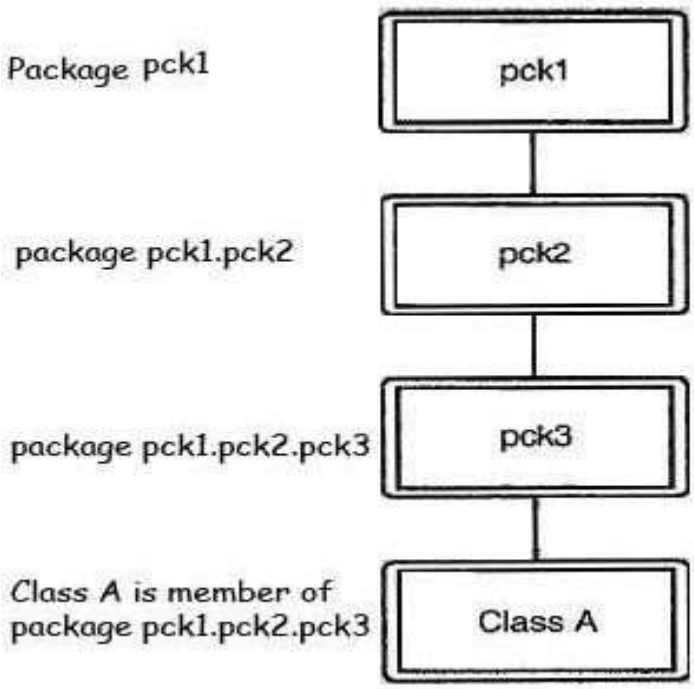
Here, *pkg* is the name of the package. For example, the following statement creates a package called MyPackage.

package MyPackage;

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**.

It is possible to create a hierarchy of packages. The general form of a multileveled package statement is shown here:

package pkg1[.pkg2[.pkg3]];



A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

package java.awt.image;

needs to be stored in **java\awt\image** in a Windows environment. We cannot rename a package without renaming the directory in which the classes are stored.

Finding Packages and CLASSPATH

First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable. Third, you can use the **- classpath** option with **java** and **javac** to specify the path to your classes.

consider the following package specification:

package MyPack

In order for a program to find **MyPack**, one of three things must be true. Either the program can be executed from a directory immediately above **MyPack**, or the **CLASSPATH** must be set to include the pathto **MyPack**, or the **-classpath** option must specify the path to **MyPack** when the program is run via **java**.

When the second two options are used, the class path *must not* include **MyPack**, itself. It must simply specify the *path to* **MyPack**. For example, in a Windows environment, if the path to **MyPack** is

C:\MyPrograms\Java\MyPack

then the class path to **MyPack** is

C:\MyPrograms\Java

Example:

```
// A simple package
package MyPack;
class Balance {
String  name;
double bal;
Balance(String n, double b) {
name = n;
```

```
bal = b;
}
void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
class AccountBalance {
public static void main(String args[]) {
Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
for(int i=0; i<3; i++) current[i].show();
}
}
```

Call this file **AccountBalance.java** and put it in a directory called **MyPack**.

Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Then, try executing the **AccountBalance** class, using the following command line:

java MyPack.AccountBalance

Remember, you will need to be in the directory above **MyPack** when you execute this command. (Alternatively, you can use one of the other two options described in the preceding section to specify the path **MyPack**.)

As explained, **AccountBalance** is now part of the package **MyPack**. This means that it cannot be executed by itself. That is, you cannot use this command line:

java AccountBalance

AccountBalance must be qualified with its package name.

Example:

```
package pck1;
class Student
```

```
{
        private int rollno;
        private String name;
        private String address;
        public Student(int rno, String sname, String sadd)
    {
          rollno = rno;
          name = sname;
          address = sadd;
     }
        public void showDetails()
     {
        System.out.println("Roll No :: " + rollno);
        System.out.println("Name :: " + name);
        System.out.println("Address :: " + address);
     }
}
     public class DemoPackage
  {
        public static void main(String ar[])
    {
        Student st[]=new Student[2];
        st[0] = new Student (1001,"Alice", "New York");
        st[1] = new Student(1002,"BOb","Washington");
        st[0].showDetails();
        st[1].showDetails();
     }
   }
```

There are two ways to create package directory as follows:

1. Create the folder or directory at your choice location with the same name as package name. After compilation of copy *.class* (byte code file) file into this folder.

2. Compile the file with following syntax.

javac -d <target location of package> sourceFile.java

The above syntax will create the package given in the *sourceFile* at the <*target location of pacakge*> if it is not yet created. If package already exist then only the *.class* (byte code file) will be stored to the package given in *sourceFile.*

Steps to compile the given example code:

Compile the code with the command on the command prompt.

javac -d DemoPackage.java

1. The command will create the package at the current location with the name pck1, and contains the file DemoPackage.class and Student.class
2. To run write the command given belowjavapckl.DemoPackage

Note: The DemoPackate.class is now stored in pck1 package. So that we've to use *fully qualified typename* to run or access it.

Output:

Roll No :: 1001 Name :: Alice Address :: New YorkRoll No :: 1002 Name :: Bob

Address :: Washington

CHAPTER FIVE

5. OOP CONCEPTS

5.1. Encapsulation

Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

Encapsulation in java is a process of wrapping code and data *together into a single unit*, for example capsule i.e. mixed of several medicines.


Capsule

We can create a fully encapsulated class in java by making all the data members ofthe class private. Now we can use setter and getter methods to set and get the data in it.

It is the **technique of making the fields in a class private and providing access to the fields via public methods**. If a field is declared private, it cannot beaccessed by anyone outside the class, thereby hiding the fields within the class. Forthis reason, encapsulation is also referred to as **data hiding.**

Encapsulation can be described as a **protective barrier that prevents the code and data being randomly accessed by other code defined outside the class**. Access to the data and code is tightly controlled by an interface.

Advantage of Encapsulation in java

- ✓ The fields of a class can be made read-only or write-only.
- ✓ A class can have total control over what is stored in its fields.
- ✓ The users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code.

Example:

Let us look at an example that depicts encapsulation:

```
/* File name : EncapTest.java */
public class EncapTest{

   private String name;
   private String idNum;
   private int age;

   public int getAge(){
      return age;
   }

   public String getName(){
      return name;
   }

   public String getIdNum(){
      return idNum;
   }

   public void setAge( int newAge){age =
      newAge;
   }

   public void setName(String newName){name
      = newName;
   }

   public void setIdNum( String newId){
      idNum = newId;
   }
}
```

The public methods are the access points to this class' fields from the outside javaworld. Normally, these methods are referred as getters and setters. Therefore any class that wants to access the variables should access them through these gettersand setters.

The variables of the EncapTest class can be accessed as below:

```
/* File name : RunEncap.java */
public class RunEncap{

   public static void main(String args[]){
      EncapTest encap = new EncapTest();
      encap.setName("James");
      encap.setAge(20);
      encap.setIdNum("12343ms");

      System.out.print("Name : " + encap.getName()+" Age :
                           "+ encap.getAge());
    }
}
```

This would produce the following result:

```
Name : James Age : 20
```

5.2. Inheritance

Inheritance can be defined as the process where one object acquires the properties of another. With the use of inheritance the information is made manageable in a hierarchical order.

When we talk about inheritance, the most commonly used keyword wouldbe **extends** and **implements**. These words would determine whether one object IS-A type of another. By using these keywords we can make one object acquire the properties of another object.

IS-A Relationship:

IS-A is a way of saying : This object is a type of that object. Let us see howthe **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 aretrue:

- ✓ 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 use of the extends keyword the subclasses will be able to inherit all theproperties of the super class except for the private properties of the super class.

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

Example:

```
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 would 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.

The **implements** keyword is used by classes to inherit from interfaces. Interfacescan never be extended by the classes.

Example:

```
public interface Animal {}

public class Mammal implements Animal{
}

public class Dog extends Mammal{
}
```

The instanceof Keyword:

Let us use the **instanceof** operator to check/determine whether Mammal is actuallyan 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 would produce the following result:

```
true
true
true
```

HAS-A relationship:

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.

Lets us look into an example:

```
public class Vehicle{}
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, wedo 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 implementationdetails 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.

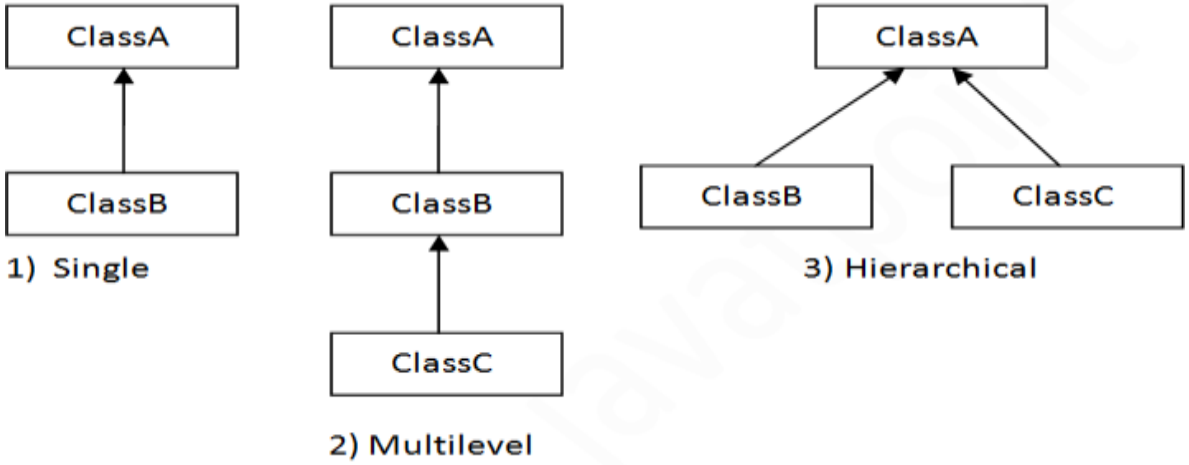
A very important fact to remember is that **Java supports only single inheritance**. This means that a class cannot extend more than one class. Therefore following is illegal:

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

However, a class can implement one or more interfaces. This has made Java get ridof the impossibility of multiple inheritances.

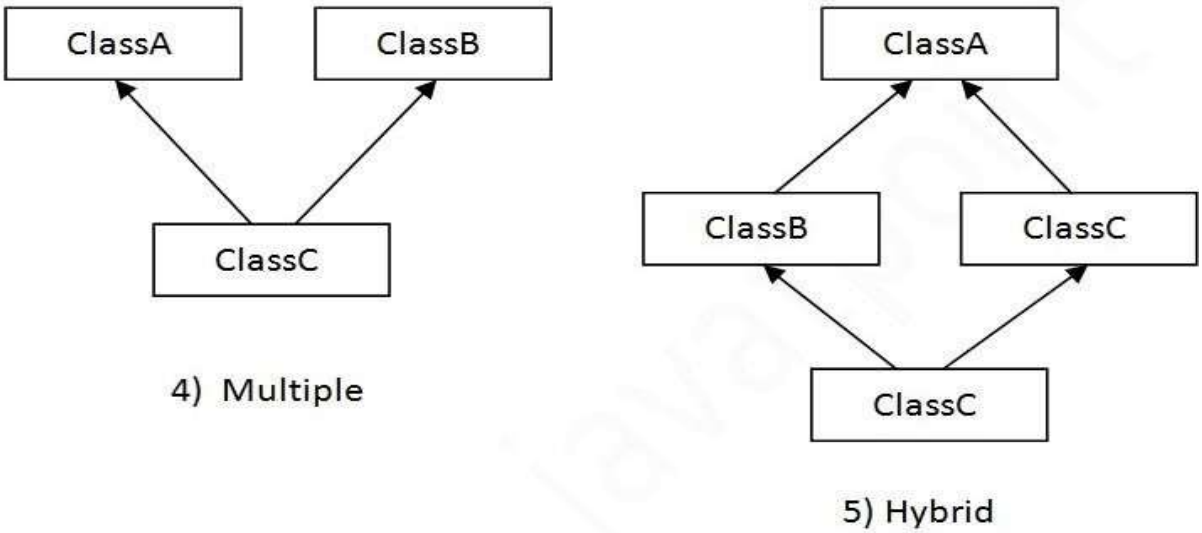
5.3. Types of Inheritance

On the basis of class, there can be three types of inheritance: single, multilevel andhierarchical in java. In java programming, multiple and hybrid inheritance is supported throughinterface only. We will learn about interfaces later.



Multiple inheritance is not supported in java in case of class.

When a class extends multiple classes i.e. known as multiple inheritance. ForExample:



.

5.4. Polymorphism

The word „polymorphism“ literally means „a state of having many shapes“ or „the capacity to take on different forms“. When applied to object oriented programminglanguages like Java, it describes a

language‟s ability to process objects of various types and classes through a single, uniform interface.

Polymorphism in Java has two types: Compile time polymorphism (static binding) and Runtime polymorphism (dynamic binding). Method overloading is an exampleof static polymorphism, while method overriding is an example of dynamic polymorphism.

An important example of polymorphism is how a parent class refers to a child classobject. In fact, any object that satisfies more than one IS-A relationship is polymorphic in nature.

For instance, let‟s consider a class Animal and let Cat be a subclass of Animal. So,any cat IS animal. Here, Cat satisfies the IS-A relationship for its own type as well as its super class Animal.

Note: It‟s also legal to say every object in Java is polymorphic in nature, as each one passes an IS-A test for itself and also for Object class.

5.4.1. Static Polymorphism:

In Java, static polymorphism is achieved through method overloading. Method overloading means there are several methods present in a class having the same name but different types/order/number of parameters.

At compile time, Java knows which method to invoke by checking the method signatures. So, this is called **compile time polymorphism** or **static binding**. The concept will be clear from the following example:

```
class DemoOverload{

    public int add(int x, int y){ //method 1return x+y;
    }

    public int add(int x, int y, int z){ //method 2return
    x+y+z;
    }

    public int add(double x, int y){ //method 3return
    (int)x+y;
    }
```

```
    public int add(int x, double y){ //method 4return
    x+(int)y;
    }

}

class Test

{

    public static void main(String[] args)

    {

    DemoOverload demo=new DemoOverload();
    System.out.println(demo.add(2,3));          //method 1 called
    System.out.println(demo.add(2,3,4));    //method    2    called
    System.out.println(demo.add(2,3.4));    //method    4    called
    System.out.println(demo.add(2.5,3));  //method 3 called
    }

}
```

In the above example, there are four versions of add methods. The first method takes two parameters while the second one takes three. For the third and fourth methods there is a change of order of parameters. The compiler looks at the method signature and decides which method to invoke for a particular method call at compile time.

5.4.2. Dynamic Polymorphism:

Suppose a sub class overrides a particular method of the super class. Let‟s say, in the program we create an object of the subclass and assign it to the super class reference. Now, if we call the overridden method on the super class reference then the sub class version of the method will be called.

Have a look at the following example. class

```
Vehicle{
        public void move(){ System.out.println("Vehicles
        can move!!");
        }
}

    class MotorBike extends Vehicle{
        public void move(){
        System.out.println("MotorBike can move and accelerate too!!");

        }
}
class Test{

        public static void main(String[] args){
        Vehicle vh=new MotorBike();
        vh.move();      // prints MotorBike can move and accelerate too!!vh=new
        Vehicle();
        vh.move();      // prints Vehicles can move!!

        }
}
```

It should be noted that in the first call to move(), the reference type is Vehicle and the object being referenced is MotorBike. So, when a call to move() is made, Java waits until runtime to determine which object is actually being pointed to by the reference. In this case, the object is of the class MotorBike. So, the move() methodof MotorBike class will be called. In the second call to move(), the object is of the class Vehicle. So, the move() method of Vehicle will be called.

As the method to call is determined at runtime, this is called **dynamic binding** or **late binding**.

What is the difference between method Overloading and Method Overriding?

There are three basic differences between the method overloading and method overriding. They are as follows:

Method Overloading	**Method Overriding**
1) Method overloading is used toincrease the readability of the program.	Method overriding is used to provide the specific implementation of the method that is already provided by its super class.
2) Method overloading is performed within a class.	Method overriding occurs in two classes that have IS-A relationship.
3) In case of method overloading parameter must be different.	In case of method overriding parameter must be same.

5.5. Abstraction

Abstraction refers to the ability to make a class abstract in OOP. An abstract classis one that cannot be instantiated. All other functionality of the class still exists, and its fields, methods, and constructors are all accessed in the same manner. You just cannot create an instance of the abstract class.

If a class is abstract and cannot be instantiated, the class does not have much use unless it is subclass. This is typically how abstract classes come about during the design phase. A parent class contains the common functionality of a collection of child classes, but the parent class itself is too abstract to be used on its own.

5.5.1. Abstract Class:

Use the **abstract** keyword to declare a class abstract. The keyword appears in the class declaration somewhere before the class keyword.

```
/* File name : Employee.java */
public abstract class Employee
{
   private String name;
   private String address;
   private int number;
   public Employee(String name, String address, int number)
   {
      System.out.println("Constructing an Employee");this.name = name;
      this.address = address;
      this.number = number;
   }
   public double computePay()
   {
     System.out.println("Inside Employee computePay");return
     0.0;
   }
   public void mailCheck()
   {
      System.out.println("Mailing a check to " + this.name
       + " " + this.address);
   }
   public String toString()
   {
      return name + " " + address + " " + number;
   }
   public String getName()
   {
      return name;
   }
   public String getAddress()
   {
```

```
      return address;
   }
   public void setAddress(String newAddress)
   {
      address = newAddress;
   }
   public int getNumber()
   {
     return number;
   }
}
```

Notice that nothing is different in this Employee class. The class is now abstract,but it still has three fields, seven methods, and one constructor.

Now if you would try as follows:

```
/* File name : AbstractDemo.java */public
class AbstractDemo
{
   public static void main(String [] args)
   {
      /* Following is not allowed and would raise error */
      Employee e = new Employee("George W.", "Houston, TX", 43);

      System.out.println("\n Call mailCheck using Employee reference--");
      e.mailCheck();
    }
}
```

When you would compile above class then you would get the following error:

```
Employee.java:46: Employee is abstract; cannot be instantiated Employee e =
      new Employee("George W.", "Houston, TX", 43);

1 error
```

Extending Abstract Class:

We can extend Employee class in normal way as follows:

```
/* File name : Salary.java */
public class Salary extends Employee
{
   private double salary; //Annual salary
   public Salary(String name, String address, int number, double salary)
   {
       super(name, address, number);
       setSalary(salary);
   }
   public void mailCheck()
   {
       System.out.println("Within mailCheck of Salary class ");
       System.out.println("Mailing check to " + getName()
       + " with salary " + salary);
   }
```

```
   public double getSalary()
   {
      return salary;
   }
   public void setSalary(double newSalary)
   {
      if(newSalary >= 0.0)
      {
         salary = newSalary;
      }
   }
   public double computePay()
   {
      System.out.println("Computing salary pay for " + getName());return
      salary/52;
   }
}
```

Here, we cannot instantiate a new Employee, but if we instantiate a new Salary object, the Salary object will inherit the three fields and seven methods from Employee.

```
/* File name : AbstractDemo.java */public
class AbstractDemo
{
   public static void main(String [] args)
   {
      Salary s = new Salary("James Gosling", "Indonesia", 3, 3600.00);Employee
      e = new Salary("John Adams", "Boston", 2, 2400.00);

      System.out.println("Call mailCheck using Salary reference --");
      s.mailCheck();

      System.out.println("\n Call mailCheck using Employee reference--");
      e.mailCheck();
    }
}
```

This would produce the following result:

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference -- Within
mailCheck of Salary class
Mailing check to James Gosling with salary 3600.0

Call mailCheck using Employee reference--Within
mailCheck of Salary class
Mailing check to John Adams with salary 2400.
```

Abstract Methods:

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as abstract.

The abstract keyword is also used to declare a method as abstract. An abstract method consists of a method signature, but no method body.

Abstract method would have no definition, and its signature is followed by a semicolon, not curly braces as follows:

```
public abstract class Employee
{
   private String name;
   private String address;
   private int number;


   public abstract double computePay();


   //Remainder of class definition
}
```

Declaring a method as abstract has two results:

- ✓ The class must also be declared abstract. If a class contains an abstract method, theclass must be abstract as well.
- ✓ Any child class must either override the abstract method or declare itself abstract.

A child class that inherits an abstract method must override it. If they do not, theymust be abstract and any of their children must override it.

Eventually, a descendant class has to implement the abstract method; otherwise,you would have a hierarchy of abstract classes that cannot be instantiated.

If Salary is extendingEmployee class, then it is required to implementcomputePay() method as follows:

```
/* File name : Salary.java */
public class Salary extends Employee
{
   private double salary; // Annual salary

   public double computePay()
   {
      System.out.println("Computing salary pay for " + getName());return
      salary/52;
   }

   //Remainder of class definition
}
```

5.5.2. Interfaces

An interface is a collection of abstract methods. A class implements an interface,thereby inheriting the abstract methods of the interface.

An interface is not a class. Writing an interface is similar to writing a class, butthey are two different concepts. A class describes the attributes and behaviors of anobject. An interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of theinterface need to be defined in the class.

An interface is similar to a class in the following ways:

- ✓ An interface can contain any number of methods.
- ✓ An interface is written in a file with a **.java** extension, with the name of theinterface matching the name of the file.
- ✓ The bytecode of an interface appears in a **.class** file.
- ✓ Interfaces appear in packages, and their corresponding bytecode file must be in adirectory structure that matches the package name.

However, an interface is different from a class in several ways, including:

- ✓ You cannot instantiate an interface.
- ✓ An interface does not contain any constructors.
- ✓ All of the methods in an interface are abstract.
- ✓ An interface cannot contain instance fields. The only fields that can appear in aninterface must be declared both static and final.
- ✓ An interface is not extended by a class; it is implemented by a class.
- ✓ An interface can extend multiple interfaces.

Declaring Interfaces:

The **interface** keyword is used to declare an interface. Here is a simple example todeclare an interface:

Example:

Let us look at an example that depicts encapsulation:

```
/* File name : NameOfInterface.java */import
java.lang.*;
//Any number of import statements

public interface NameOfInterface
{
   //Any number of final, static fields
   //Any number of abstract method declarations\
}
```

Interfaces have the following properties:

- ✓ An interface is implicitly abstract. You do not need to use the **abstract** keywordwhen declaring an interface.
- ✓ Each method in an interface is also implicitly abstract, so the abstract keyword isnot needed.
- ✓ Methods in an interface are implicitly public.

Example:

```
/* File name : Animal.java */
interface Animal {

   public void eat();
   public void travel();
}
```

Implementing Interfaces:

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

```
/* File name : MammalInt.java */
public class MammalInt implements Animal{

   public void eat(){
      System.out.println("Mammal eats");
   }

   public void travel(){
      System.out.println("Mammal travels");
   }

   public int noOfLegs(){
      return 0;
   }

   public static void main(String args[]){
      MammalInt m = new MammalInt();
      m.eat();
      m.travel();
   }
   }
```

This would produce the following result:

```
Mammal eats
Mammal travels
```

When overriding methods defined in interfaces there are several rules to be followed:

- ✓ Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
- ✓ The signature of the interface method and the same return type or subtype shouldbe maintained when overriding the methods.
- ✓ An implementation class itself can be abstract and if so interface methods need notbe implemented.

When implementation interfaces there are several rules:

- ✓ A class can implement more than one interface at a time.
- ✓ A class can extend only one class, but implement many interfaces.
- ✓ An interface can extend another interface, similarly to the way that a class canextend another class.

Extending Interfaces:

An interface can extend another interface, similarly to the way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

The following Sports interface is extended by Hockey and Football interfaces.

```
//Filename: Sports.java
public interface Sports
{
   public void setHomeTeam(String name); public
   void setVisitingTeam(String name);
}
//Filename: Football.java
public interface Football extends Sports
{
   public void homeTeamScored(int points); public
   void visitingTeamScored(int points);public void
   endOfQuarter(int quarter);
}
//Filename: Hockey.java
public interface Hockey extends Sports
{
   public void homeGoalScored(); public
   void visitingGoalScored(); public void
   endOfPeriod(int period);public void
   overtimePeriod(int ot);
}
```

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football andthe two methods from Sports.

Extending Multiple Interfaces:

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as:

```
public interface Hockey extends Sports, Event
```

CHAPTER SIX

6. EXCEPTION HANDLING

Exceptions-exception hierarchy-throwing and catching exceptions-built-in exceptions, creating own exceptions, Stack Trace Elements.

6.1. Difference between error and exception

Errors indicate serious problems and abnormal conditions that most applicationsshould not try to handle. Error defines problems that are not expected to be caught under normal circumstances by our program. For example memory error, hardware error, JVM error etc.

Exceptions are conditions within the code. A developer can handle such conditionsand take necessary corrective actions. Few examples

- ✓ DivideByZero exception
- ✓ NullPointerException
- ✓ ArithmeticException
- ✓ ArrayIndexOutOfBoundsException

An exception (or exceptional event) is a problem that arises during the execution of a program.

When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

If an exception is raised, which has not been handled by programmer then program execution can get terminated and system prints a non user friendly error message.

Ex: Exception in thread "main" java.lang.ArithmeticException: / by zero at ExceptionDemo.main(ExceptionDemo.java:5)

Where, ExceptionDemo : The class name
main : The method name
ExceptionDemo.java : The
filenamejava:5 : Line number

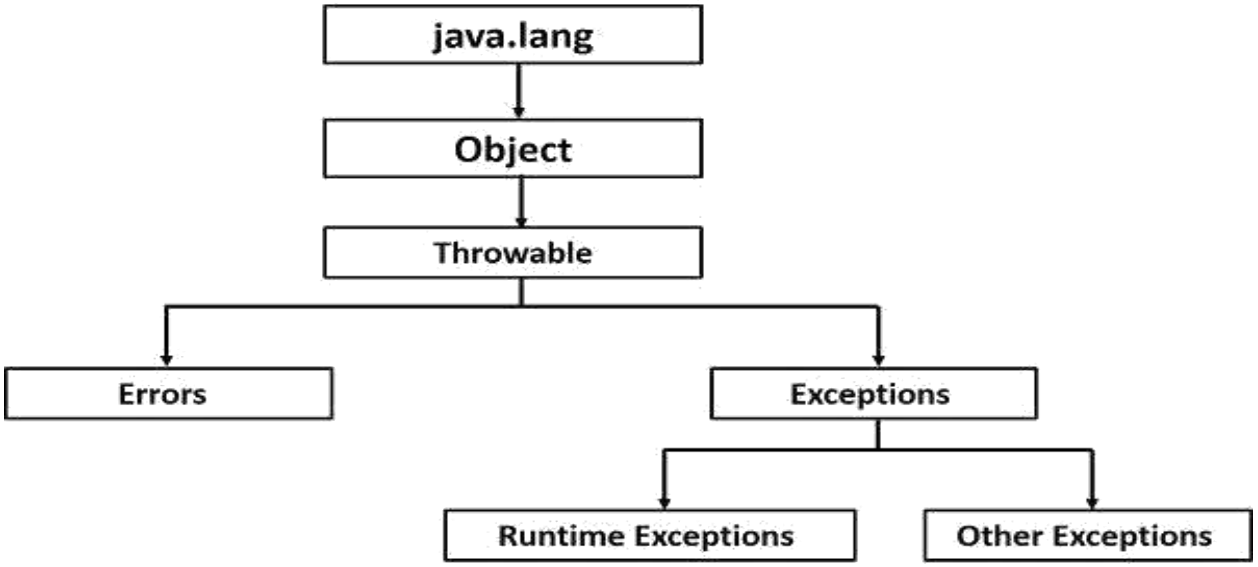
An exception can occur for many different reasons. Following are somescenarios where an exception occurs.

- ✓ A user has entered an **invalid data.**
- ✓ A file that needs to be opened cannot be found.

- ✓ A network connection has been lost in the middle of communications orthe JVM hasrun out of memory.

6.2. Exception Hierarchy

All exception classes are subtypes of the java.lang.Exception class. Theexception class is a subclass of the Throwable class.



Key words used in Exception handling

There are 5 keywords used in java exception handling.

1. try	A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code.
2. catch	A catch statement involves declaring the type of exception we are trying to catch.
3. finally	A finally block of code always executes, irrespective of occurrence of an Exception.
4. throw	It is used to execute important code such as closing connection, stream etc. throw is used to invoke an exception explicitly.
5. throws	throws is used to postpone the handling of a checked exception.

Syntax :

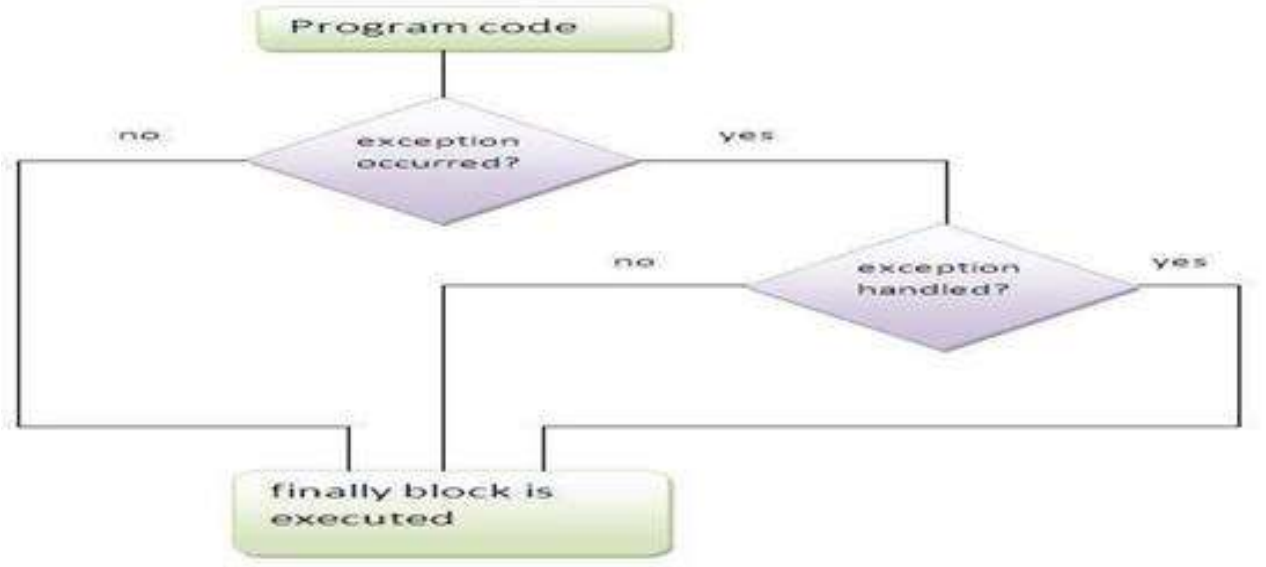
```
try
{
    //Protected code
}

catch(ExceptionType1 e1)
{

    //Catch block
}
catch(ExceptionType2 e2)
{
    //Catch block
}
    catch(ExceptionType3 e3)

{

    //Catch block
}
finally
{

//The finally block always
executes.
}
```

```
//Example-predefined Excetion - for
//ArrayindexoutofBounds Exception
public class ExcepTest
{
public static void main(String args[])
 { int a[] = new int[2];
      try
      { System.out.println("Access element three :" +
      a[3]);
        }
          catch(ArrayIndexOutOfBoundsException e)
          { System.out.println("Exception thrown :" + e);
        }
      finally
       { a[0] = 6;
        System.out.println("First element value: " + a[0]);
    System.out.println("The finally statement is
    executed");
      }
   }
}
```

Output

```
Exception thrown
:java.lang.ArrayIndexOutOfBoundsException:3
First element value: 6
The finally statement is executed
```

Note : here array size is 2 but we are trying to access 3rdelement.



6.3. Uncaught Exceptions

This small program includes an expression that intentionally causes a divide-by- zero error:

```
class Exc0

{

public static void main(String args[]) {

 int d = 0; int a =42 / d;

 }

 }
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of Exc0 to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.

Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program. Here is the exception generated when this example is executed:

```
java.lang.ArithmeticException: / by zero at Exc0.main(Exc0.java:4)
```

Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program. Here is the exception generated when this example is executed:

java.lang.ArithmeticException: / by zero at Exc0.main(Exc0.java:4)

Using try and Catch

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, includea catch clause that specifies the exception type that you wish to catch. A try and its catch statement form a unit. The the following program includes a try block and a catch clause that processes the ArithmeticException generated by the division-by-zero error:

```
class Exc2 {

public static void main(String args[]) {int d, a;

try { // monitor a block of code.d
= 0;

a = 42 / d;

System.out.println("This will not be printed.");

} catch (ArithmeticException e) { // catch divide-by-zero error
System.out.println("Division by zero.");

}

System.out.println("After catch statement.");

}

}
```

This program generates the following output:

Division by zero.

After catch statement.

The call to println( ) inside the try block is never executed. Once an exception isthrown, program control transfers out of the try block into the catch block.

Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block.

The following example traps two different exception types:

```
// Demonstrate multiple catch statements.

class MultiCatch {

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

int a = args.length; System.out.println("a = " + a);int b = 42 / a;

int c[] = { 1 };c[42] = 99;

} catch(ArithmeticException e) { System.out.println("Divide by 0: " + e);

} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index oob: " + e);

}

System.out.println("After try/catch blocks.");

}

}
```

Here is the output generated by running it both ways:

```
C:\>java MultiCatch
a = 0

Divide by 0: java.lang.ArithmeticException: / by zeroAfter
try/catch blocks.
```

C:\>java MultiCatch TestArga
= 1

Array index oob: java.lang.ArrayIndexOutOfBoundsException:42After
try/catch blocks.

Nested try Statements

The try statement can be nested. That is, a try statement can be inside the block of another try. Each time a try statement is entered, the context of that exceptionis pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception.

```
// An example of nested try statements.
class NestTry {

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

int a = args.length;

/* If no command-line args are present,the
following statement will generate

a divide-by-zero exception. */
int b = 42 / a;
System.out.println("a = " + a);
try { // nested try block

/* If one command-line arg is used,then a
divide-by-zero exception

will be generated by the following code. */if(a==1)
a = a/(a-a); // division by zero

/* If two command-line args are used,

then generate an out-of-bounds exception. */
if(a==2) {
```

```
int c[] = { 1 };

c[42] = 99; // generate an out-of-bounds exception

}

} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index out-of-bounds: " + e);

}

} catch(ArithmeticException e) { System.out.println("Divide by
0: " + e);

}

}

}
```

```
C:\>java NestTry

Divide by 0: java.lang.ArithmeticException: / by zeroC:\>java
NestTry One

a = 1

Divide by 0: java.lang.ArithmeticException: / by zeroC:\>java
NestTry One Two

a = 2

Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException:42
```

6.4. throw

It is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here:

throw ThrowableInstance;

Here, ThrowableInstance must be an object of type Throwable or a subclass ofThrowable.

Primitive types, such as int or char, as well as non-Throwable classes, such as String and

Object, cannot be used as exceptions. There are two ways you can obtain a Throwable object: using a parameter in a catch clause, or creating one with the new operator.

The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace

```
// Demonstrate
throw. class
ThrowDemo { static
void demoproc() {try
{

throw new NullPointerException("demo");
} catch(NullPointerException e) {
System.out.println("Caught inside
demoproc.");throw e; // rethrow the
exception

}
}
```

```
public static void main(String
args[]) {try {
demoproc();
} catch(NullPointerException e) {
System.out.println("Recaught: " +
e);
}}}
```

Here is the resulting output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

6.5. Throws

If a method is capable of causing an exception that it does not handle, it must specify this behaviour so that callers of the method can guard themselves against that exception. You do this by including a throws clause in the method'sdeclaration. A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses. All other exceptions that a method can throw must be declared in the throws clause. This is the general form of a method declaration that includes a throws clause:

```
type method-name(parameter-list) throws exception-list
{
// body of method
}
```

Here, exception-list is a comma-separated list of the exceptions that a methodcan throw.

```
class ThrowsDemo {
```

```
static void throwOne() throws IllegalAccessException
{System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String
args[]) {try {
throwOne();
} catch (IllegalAccessException e) {
System.out.println("Caught " + e);
}
}
}
```

Here is the output generated by running this example program:inside throwOne

caught java.lang.IllegalAccessException: demo

6.6. finally

The finally keyword is designed to address this contingency. finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. Any time amethod is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The finally clause is optional.

```
// Demonstrate
finally.class
FinallyDemo {

// Through an exception out of the
method.static void procA() {

try {
System.out.println("inside procA");
throw new
RuntimeException("demo");

} finally {
System.out.println("procA's finally");
}
}
// Return from within a try
block.static void procB() {

try {
System.out.println("inside
procB");return;

} finally {
System.out.println("procB's
finally");

}
}
```

```
// Execute a try block
normally.static void procC()
{
try {
System.out.println("inside procC");
} finally {
System.out.println("procC's
finally");
}
}
public static void main(String
args[]) {try {
procA();
} catch (Exception e) {
System.out.println("Exception
caught");
}
procB();
procC();
}
}
```

Here is the output generated by the preceding program:

```
inside procA
procA’s
finally
Exception
caughtinside
```

procB

procB’s

finally inside

procC

procC’s

finally

6.7. Types of exceptions

Checked exceptions −A checked exception is an exception that occurs at the compiletime, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

Unchecked exceptions − An unchecked exception is an exception that occurs at thetime of execution. These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

Common scenarios where exceptions may occur:

There are given some scenarios where unchecked exceptions can occur.They are as follows:

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.int a=50/0;//ArithmeticException

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by thevariable occurs an NullPointerException.

```
String s=null;
System.out.println(s.length());//NullPointerException
```

3) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would resultArrayIndexOutOfBoundsException as shown below:

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

6.8. User-defined Exceptions

All exceptions must be a child of **Throwable.**

If we want to write a checked exception that is automatically enforced bythe Handle ,we need to extend the Exception class.

User defined exception needs to inherit (extends) Exception class in order to act as an exception.

throw keyword is used to throw such exceptions.

```
class MyOwnException extends Exception
    {
    public MyOwnException(String msg) {
    super(msg);
        }
    }
    class EmployeeTest
```

```
{
        static void employeeAge(int age) throws
        MyOwnExceptio
          n {if(age < 0)
            throw new MyOwnException("Age can't be less
          than zero"); else
            System.out.println("Input is valid!!");
}
  public static void main(String[] args)
      {
          try { employeeAge(-2);
              }
          catch (MyOwnException e)
          {
              e.printStackTrace();
            }
 }
}
```

Advantages of Exception Handling

Exception handling allows us to control the normal flow of the program by using exception handling in program.

It throws an exception whenever a calling method encounters an error providing that the calling method takes care of that error.

It also gives us the scope of organizing and differentiating between different error types using a separate block of codes. This is done with thehelp of try-catch blocks.

CHAPTER SEVEN

7. JAVA APPLETS

7.1. What is applet?

An applet is a program written in the Java programming language that can be included in an HTML page, much in the same way an image is included in a page.

- ✓ An applet is a Java program designed to be executed via a Java-enabled web browser.
- ✓ When you use a Java technology enabled browser to view a page that contains an applet, the applet's code is transferred to your system and executed by the browser's Java Virtual Machine (JVM).
- ✓ Applet programs can be viewed from web browsers such Internet Explorer and Netscape, Google Chrome etc.
- ✓ A special Java class that doesn't run in a DOS window.

What Java Applets Can do?

- ✓ Display message and image on a web page
- ✓ Draw picture on a web page
- ✓ Receive input from the user through keyboard or mouse
- ✓ Play Sound

Advantage of Applet

- ✓ It works at client side so less response time
- ✓ It can be executed by any browsers that has JVM under many platforms, including Linux, Windows, Mac Os etc.

Drawback of applet

- ✓ Plugin is required at client browser to execute applet

Applications and applets

- ✓ Java applications are executed from the command line
- ✓ A Java JVM must be installed
- ✓ The JVM is given the name of a class which contains a main() method
- ✓ The main method instantiates the objects necessary to start the application
- ✓ They are given more security
- ✓ They are allowed to read and write to local file system

Applets are executed by a browser

- ✓ The browser either contains a JVM or loads the Java plugin
- ✓ The programmer must implement a class which is a subclass of applet
- ✓ There is no main method. Instead, the applet contains an init() method which is invoked by the Browser when the applet starts
- ✓ Applets interact with user through **awt**, not through console based I/O classes.
- ✓ Applets cannot read from or write to local file system.
- ✓ Applets interact with the user while the web page is active and stop their execution when his web page is no longer active.

How to run an applet?

There are two ways to run an applet?

a) By html file(using website browser)

b) By appletviewer tool(for testing purpose)

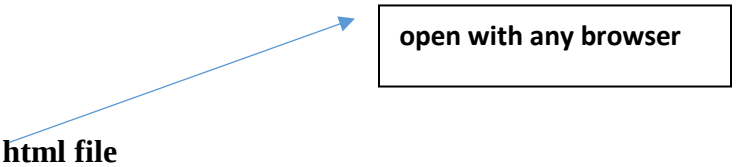
a) Simple example of applet by html file to display 'Welcome' text

- ✓ Step 1: to execute the applet by html file, create an applet code and **compile** it.
- ✓ Step 2: after that create an html file and place the applet code in html file.
- ✓ Step3 : open the html file by any browser

Case 1: By html file(using website browser)

//save as **First.java**

```
import java.awt.Graphics;
import java.applet.Applet;
public class First extends Applet
{
public void paint(Graphics g)
{
g.drawString("Welcome",50,80);
}
}
```



// save as **Myapplet.html-----→name of the html file**

```
<html>
<body>
<applet code="First.class" width=500 height=300>
</applet>
</body>
</html>
```

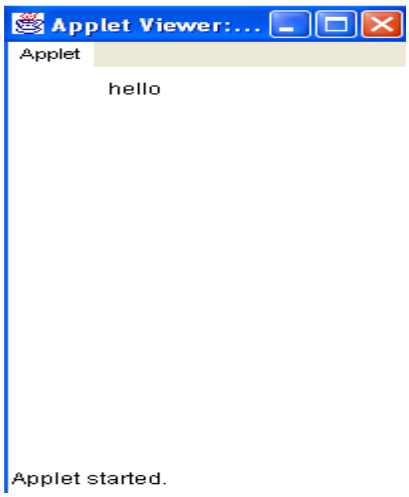
Case 2: by appletviewer

Step 1: create an applet that contains applet tag in comment and compile it.

Step 2: after that run it by appletviewer

Save as First.java

```
import java.awt.*;
import java.applet.*;
/*
<applet code="First.class" width=200 height=300>
</applet>
*/
public class First extends Applet
{
public void paint(Graphics g)
{
g.drawString("hello",50,50);
}
}
```



To compile and execute, open cmd and type

javac First.java

appletviewer First.java

Note :

- ✓ An applet executes within an environment provided by **web browser** or a tool such as the **appletviewer**.
- ✓ Therefore it doesn't have a main() Method.
- ✓ **Appletviewer** is generally used by developers for testing their applets before deploying them to a website.
- ✓ An applet viewer executes your applet in a **window**.

7.2. The Applet class

- ✓ To create an applet, you must import the Applet class
- ✓ This class is in the java.applet package
- ✓ The Applet class contains code that works with a browser to create a display window
- ✓ *Capitalization matters!*

  applet and Applet are different names

Importing the Applet class

Here is the directive that you need: import java.applet.Applet;

- ✓ import is a keyword
- ✓ java.applet is the name of the package
- ✓ A dot (.) separates the package from the class
- ✓ Applet is the name of the class
- ✓ There is a semicolon (;) at the end

The java.awt package

- ✓ "awt" stands for "Abstract Window Toolkit"
- ✓ The java.awt package includes classes for:
- ✓ Drawing lines and shapes
- ✓ Drawing letters
- ✓ Setting colors
- ✓ Choosing fonts

- ✓ If it's drawn on the screen, then java.awt is probably involved!

Importing the java.awt package

- ✓ Since you may want to use many classes from the java.awt package, simply import them all:
- ✓ import java.awt.*;
- ✓ The asterisk, or star (*), means "all classes"
- ✓ The import directives can go in any order, but must be the first lines in your program

Your applet class

- ✓ public class Drawing extends Applet { … }
- ✓ Drawing is the name of your class
- ✓ Class names should always be capitalized
- ✓ extends Applet says that our Drawing is a kind of Applet, but with added capabilities
- ✓ Java's Applet just makes an empty window
- ✓ We are going to draw in that window
- ✓ The *only* way to make an applet is to extend Applet

Applet Display Methods

- ✓ Applet uses **awt** to perform input/output .
- ✓ To output a string to an applet we use **drawstring().**
- ✓ It is a member of Graphic class. It should be called from either **paint() or update()** methods.
- ✓ Syntax: **void drawstring(String *message*, int *x*, int *y*)**
- ✓ Here *message* specifies the String to be displayed and '***x***' and '***y***' specifies the location where the string is to be displayed.
- ✓ (**0**, **0**) is upper left corner of screen

The paint method

- ✓ Our applet is going to have a method to paint some colored rectangles on the screen
- ✓ This method must be named paint
- ✓ paint needs to be told where on the screen it can draw
- ✓ This will be the only parameter it needs
- ✓ paint doesn't return any result
- ✓ public void paint(Graphics g) { … }
- ✓ public says that anyone can use this method
- ✓ void says that it does not return a result
- ✓ A Graphics (short for "Graphics context") is an object that holds information about a painting
- ✓ It remembers what color you are using
- ✓ It remembers what font you are using
- ✓ You can "paint" on it (but it doesn't remember what you have painted)

7.3. Life Cycle of an Applet

- ✓ Most applets override these methods
- ✓ Applets are executed when life cycle methods are called by JVM

a) Applet is initialized

b) Applet is started

c) Applet is painted

d) Applet is stopped

e) Applet is destroyed

a) init( )

- ✓ Init () method begins the applet life cycle
- ✓ This is where we should initialize variables and methods.

- ✓ Its is called only once in the life cycle of the applet

b) Start( )

- ✓ Start method is called after the init( )
- ✓ This method is called to restart an applet after it has been stopped(or maximized)
- ✓ User can interact within the applet
- ✓ It can call any number of times
- ✓ Every applet that does something after initialization must override start() method.

c) stop( )

- ✓ It can be called any no of times like start()
- ✓ To stop the applet's execution such as when the user leaves the applet's page or quits the browser or
- ✓ When the page is Quit or the Applet Viewer is minimized.
- ✓ Most applets that override start method should also override the stop method.
- ✓ At least once in the life cycle

d) destroy()

- ✓ It is called once like init(), when Quitting the applet viewer.
- ✓ Destroy() method is called when your applet needs to be removed completely from memory
- ✓ Most applet's don't need to override the destroy method, since their stop method does everything to shut down the applet's execution.
- ✓ However destroy is available for applet's that need to release additional resources.

- ✓ When an applet begins, the AWT calls the following methods, in this sequence:
  - **init( )**
  - **start( )**
  - **paint( )**
- ✓ When an applet is terminated, the following sequence of method calls takes place:
  - **stop( )**
  - **destroy( )**

Program to illustrate the working of life cycle methods

```
import java.awt.*;
import java.applet.*;
/*<applet code="LifeCycle" width=300 height=300>
</applet>*/
public class LifeCycle extends Applet
{
String str="";
public void init()
{
str+="init";
}
public void start()
{
str+="start"
}
public void stop()
{
str+="stop"
}
public void destroy()
{
```

```
System.out.println("destroy") ;
}
public void paint(Graphics g)
{
g.drawString(str,10,25);
} }
```

When this applet first begins, you can see it displays init;start.

- ✓ If you minimize and maximize the applet window, you can observe that the applet then displays init; start; stop; start.
- ✓ This is because the stop() method was called when the applet was minimized and the start() method was called when it was maximized.

7.4. Displaying Graphics in Applet

Commonly used methods of Graphics class:

1.Public abstract void drawString(String str, int x, int y):

Is used to draw the specified string

2. Public void drawRect(int x, int y,int width, int height):

Draw the rectangle with the specified width and height

3. Public abstract void fillRect(int x, int y, int width, int height):

Is used to fill rectangle with the default color

4. Public abstract void drawOval(int x, int y, int width, int height):

Is used to draw oval with the specified width and height

Public abstract void drawLine(int x, int y, int x2, int y2):

Is used to draw line between the points(x1,y1) and (x2,y2)

6. Public abstract setColor(Color c):

Is used to set the graphics current color to the specified color.

7. Public abstract void setFont(Font font):

Is used to set the graphics current font to the specified font.

Colors

- ✓ The java.awt package defines a class named Color
- ✓ There are 13 predefined colors—here are their fully-qualified names:

- ✓ Java also allows color names in lowercase letters.

Example

```
import java.applet.Applet; import java.awt.*;
public class Drawing extends Applet {
public void paint(Graphics g) {
g.setColor(Color.BLUE); g.fillRect(20, 20, 50, 30);
g.setColor(Color.RED); g.fillRect(50, 30, 50, 30); } }
/*
<applet code="Drawing" width=200 height=300>
</applet>
*/
```