Preface

This book is released for quick reference of programming language Java i.e.., Object Oriented Programming. Programming acts as an interface between the computer and the user. There are many programming languages in that JAVA is one among plays a major role now-a-days for web application development and gaming.

This book provides the complete knowledge of core java. The Quick Reference JAVA is useful for learning java in practical oriented way as well as for examinations. This book covers the syllabi in the curriculum of various universities in India and abroad.

This book is organized as follows. It consists of twelve chapters.

Chapter 1 provides the introduction to Java covering history, Real-time applications and comparison with previous programming languages like c and c++.

Chapter 2 introduces the concepts of object oriented thinking which is really necessary in java and is most famous concepts, also called pillars of java that is OOP's concepts.

Chapter 3 covers all basic concepts of programming ranging from variable declaration to control statements and many other concepts used in java programming.

Chapter 4 explains in brief about inheritance concept in java and various types of it with example programs and diagrams.

Chapter 5 introduces the concepts of packages and interfaces which are mostly used in java programming are explained in detail.

Chapter 6 discusses the various techniques of Exceptional Handling in java programming and its implementation with example programs.

Chapter 7 discusses the concept of Multithreading, how multiple tasks can be done at an instance of time.

Chapter 8 presents the basics of input and output streams, only important and frequently used predefined packages, classes and methods used in java are explained in detail.

I would like to express my sincere thanks to all the contributors for their much needed cooperation, authoritative and up to date information organized in a befitting manner.

I extend my sincere thanks to C. Vishnu Vardhan Reddy, Manchi Jayaprakash, C. Rachana, Dr. M. Vijaya Bhasakar Reddy, and P. SasiKala.

I extend my sincere thanks to Manchi Thej Kumar, Thanooja Kotika, Manasa Uggurarapu, Sandraneni Jyoshna, Thalari Anitha, Sujith Kumar Peta, my colleagues, and the publisher LAP for their valuable support for facilitating in completion of this volume.

The first edition of the textbook was aimed to elucidate the quick reference of Java, especially designed and written for the purpose of Internal & External Exams for their Academics of all Regulations of Various Universities Curriculum.

Dr. C. Shoba Bindu
Mr. P. Dileep Kumar Reddy
Dr. K. Dhana Sree Devi

INDEX

CONTENTS

CHAPTER IV: INHERITANCE

Introduction, Types of Inheritance, forms of inheritance, benefits of inheritance, costs of inheritance, Member access rules, using super and final with inheritance, polymorphism method overriding, abstract classes.

CHAPTER V: PACKAGES AND INTERFACES

Packages - Introduction, Defining, Creating and Accessing a Package, Understanding CLASSPATH, importing packages.

Interfaces- Differences between classes and interfaces, Defining an interface, Implementing interface, Applying interfaces, variables in interface and extending interfaces.

CHAPTER VI: EXCEPTION HANDLING

Introduction to Exception handling Fundamentals, Exception Types, Uncaught Exceptions, benefits of exception handling, Termination or resumptive models, exception hierarchy, usage of try, catch, throw, throws and finally, java built in exceptions, Creating your own exception subclasses, Chained Exceptions, Three Recently added Exceptions features.

CHAPTER VII: MULTITHREADING

Fundamentals, The java Thread Model, The main thread, Creating Thread, Thread class, Runnable interface, creating multiple threads, Life cycle of thread, Thread priorities, Using isAlive() and join(), Synchronization, Thread communication, Obtaining a thread state, difference between multithreading and multitasking.

CHAPTER VIII: I/O AND OTHER TOPICS

Introduction, I/O basics, Reading Console input, Writing console Output, Reading and writing files, Automatically closing a file, Random-access files, File I/O using character streams, Wrappers.

CHAPTER 1

INTRODUCTION TO JAVA

1.1 Java's Lineage:

Java is the extension of previous programming languages like c and c++. This programming language inherits the maximum properties from c and c++. The meaning of Java's Lineage is it is the next descendant language of c and c++. However, the concepts of object oriented programming are taken from c++. The basic concepts like variables, data types, and so on are taken from c to c++ and then to java. Likewise the properties of one language to another language have been inherited and added some more extensions or more properties to it. The creation of java was rooted in the process of adaptation and filtering from the previous computer programming languages.

1.2 The Creation of JAVA/ Evolution of JAVA:

The creation of java has taken place in 1991 by sun micro systems. James Gosling team in the company worked years together for creation of java. Now java is everywhere, in developing applications especially for android, web applications, gaming, internet, and also mostly used in business. One day in the developing team members of java are sitting together and having a cup of green tea. Beside them outside the window there is an **'oak'** tree. They have kept name initially for this programming language as **‘oak’** in 1991. The symbol/logo for java, we can see a tea cup. This is how they named as oak and kept logo as tea cup.

The developing team members are James Gosling, Chris Worth, Ed Frank, Patrick Naughton, and Mike Sherdian. To develop java for them it took nearly 18 months. And later in 1995 they renamed the programming language as ‘**JAVA**’.

Java Development Kit (JDK) Beta version is released in 1994. Then in 1996 the first version of java, jdk 1.0 was released. The following table shows version history of JAVA.

Version Name	Released Year
JDK Beta	1994
JDK 1.0	1996
JDK 1.1	1997
J2SE 1.2	1998
J2SE 1.3	2000
J2SE 1.4	2002
J2SE 5.0	2005
JAVA SE 6	2006
JAVA SE 7	2011
JAVA SE 8	2014

Now JAVA is acquired by Oracle Company from Sun Micro Systems on 27th January 2010.

In each version of java some of the features have been added. JAVA also contains three editions. They are

1. JAVA SE
2. JAVA ME
3. JAVA EE

1. JAVA SE:

Java Platform, Standard Edition or **Java SE** is most widely used platform for the development and deployment of coding for desktop and server environments. **Java SE** uses the object-oriented programming language concepts. It is one among the part of Java software-platform family.

2. JAVA ME:

Java Platform, Micro Edition or Java ME is the subset of SE and also this is especially used to develop mobile applications. Now-a-days over billions of mobile devices uses java.

3. JAVA EE:

Java Platform, Enterprise Edition or Java EE is built on top of Java SE, and it is used for developing web applications and large-scale enterprise applications and so on.

So, for developing any applications using java, JAVA SE must be installed on your computing system.

1.3 **How JAVA changed the internet**:

The Internet helped JAVA to be as first programming and JAVA itself had a great impact on the Internet. While simplifying the web programming JAVA innovated a new type of networked program called the applet that changed the way that can be used in online world thought about content. JAVA also notified about some of main issues associated with the Internet such as portability and security. Let’s have in brief about more at each of these issues.

1. **Security:** We know that whenever we download a normal program we have a risk in internet, because the code that we are downloading may have virus of any other harmful code. Malicious code can cause damage to our computing environments and also it can gain unauthorized access to system resources. In order to download the code or a program from internet and executed on the client computer safely, we need to use applets which prevents harm to our system. Such that Java achieved this protection by ensuring and enhancing the security measures in an applet to the JAVA execution environment and it doesn’t allow its access to other parts of the computer hardware or software to damage.
2. **Portability:** It is a major part of the Internet because there are many different types of computers or devices and

operating systems connected to it. Some means of generating portable executable code is necessary. For example, In case of an applet, It must be able to be downloaded and executed by the wide variety of CPU's, operating systems and browsers connected to the internet safely.

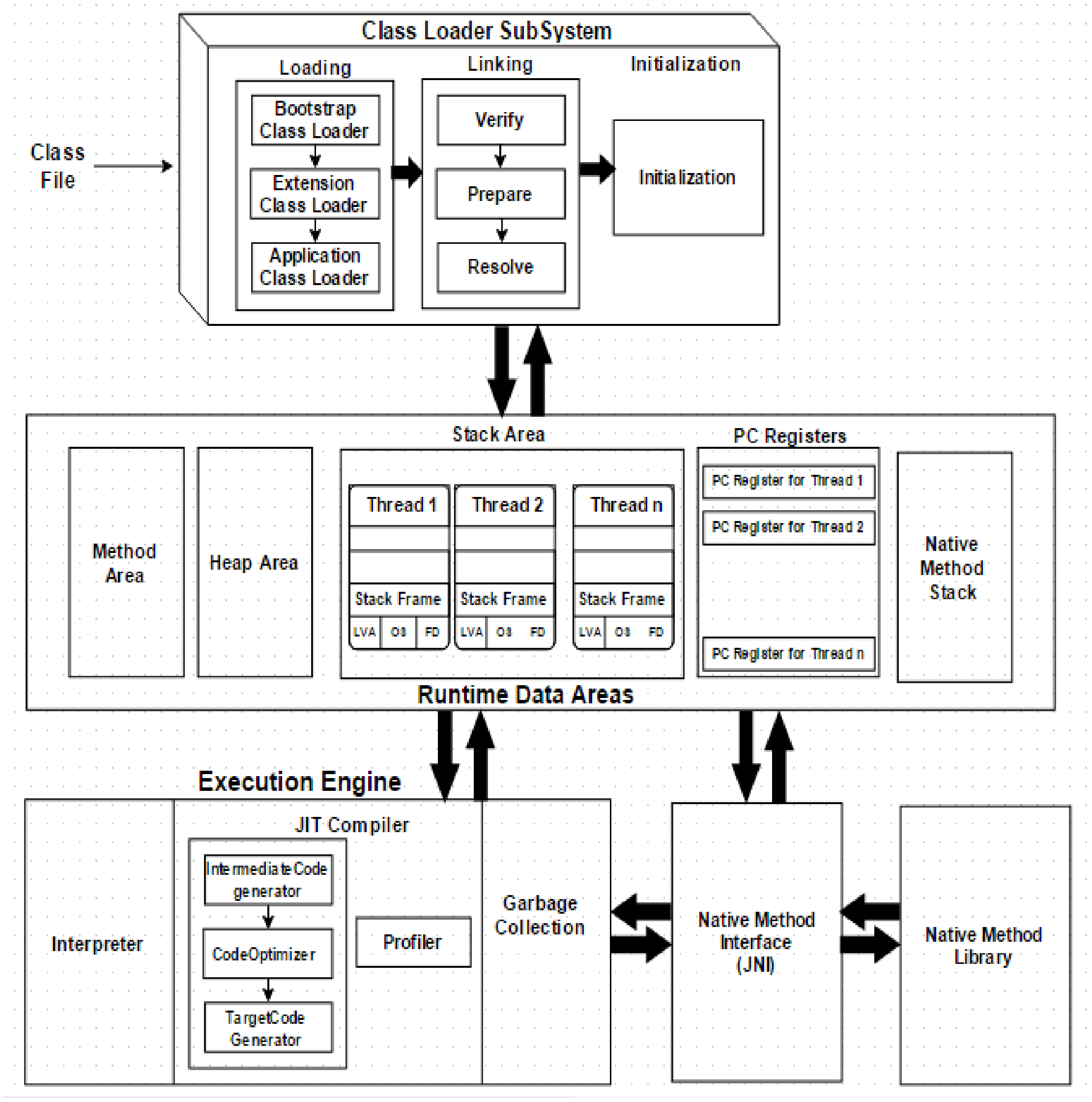
1.4 Java's Magic: The Byte Code

Java's byte code is platform independent because once a source code is compiled and executed, then java compiler generates byte code of text file. This can be taken that is copied to any other machine/computer which is having any type of operating system such as windows, Linux, UNIX, and Mac and so on; java program can be executed at any of those computing environments running a Java virtual machine (JVM). This is the reason why we call Java Byte Code is platform independent.

1.5 JVM Architecture:

Every Java developer knows that byte code will be executed by **JRE** (**Java Runtime Environment**). But many don't know the fact that **JRE** is implemented by Java Virtual Machine (JVM), which analyzes the byte code, interprets the code, and executes it

A **Virtual Machine** is a software implementation of a physical machine. Java was developed with the concept of **WORA (*Write Once Run Anywhere*)**, which runs on a **Virtual Machine**. The **compiler** compiles the Java file into a Java **.class** file, then that **.class** file is input into the JVM, which Loads and executes the class file. The following diagram depicts the Architecture of the JVM in detail.



1.6 Downloading and Installation of JAVA SE:

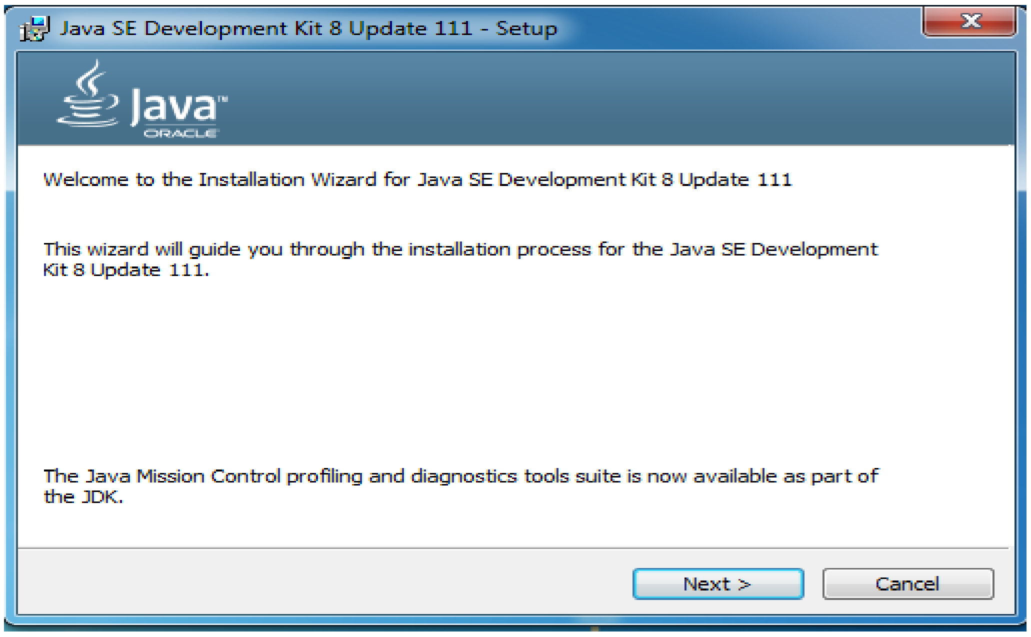
In the URL bar, type this link or simply type in google as java se download you can now see this link at the first result of search.

URL:
http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html

In that link it will show you different java development kits of both 32 and 64 bit operating system compatibility .exe files links. By accepting the license agreement, we can be able to download any of the java kits for our purpose.

Installation Process:

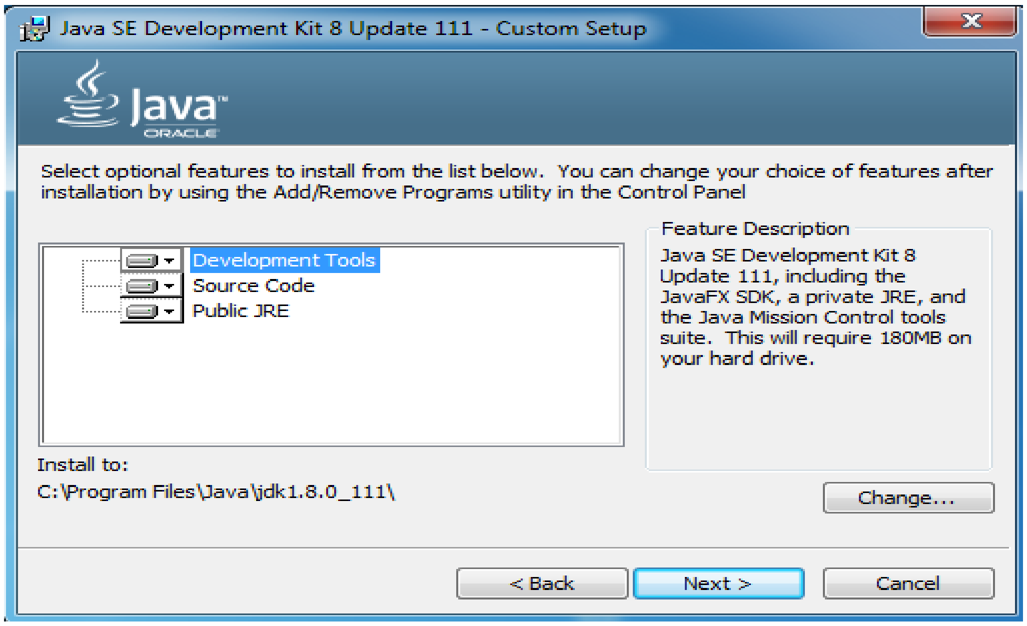
As a Computer science student can install the software very easily but remaining streams may need guidance. For that now we will have a step by step process of screenshots as follows:



Welcome to the Installation Wizard for Java SE Development Kit 8 Update 111

This wizard will guide you through the installation process for the Java SE Development Kit 8 Update 111.

The Java Mission Control profiling and diagnostics tools suite is now available as part of the JDK.

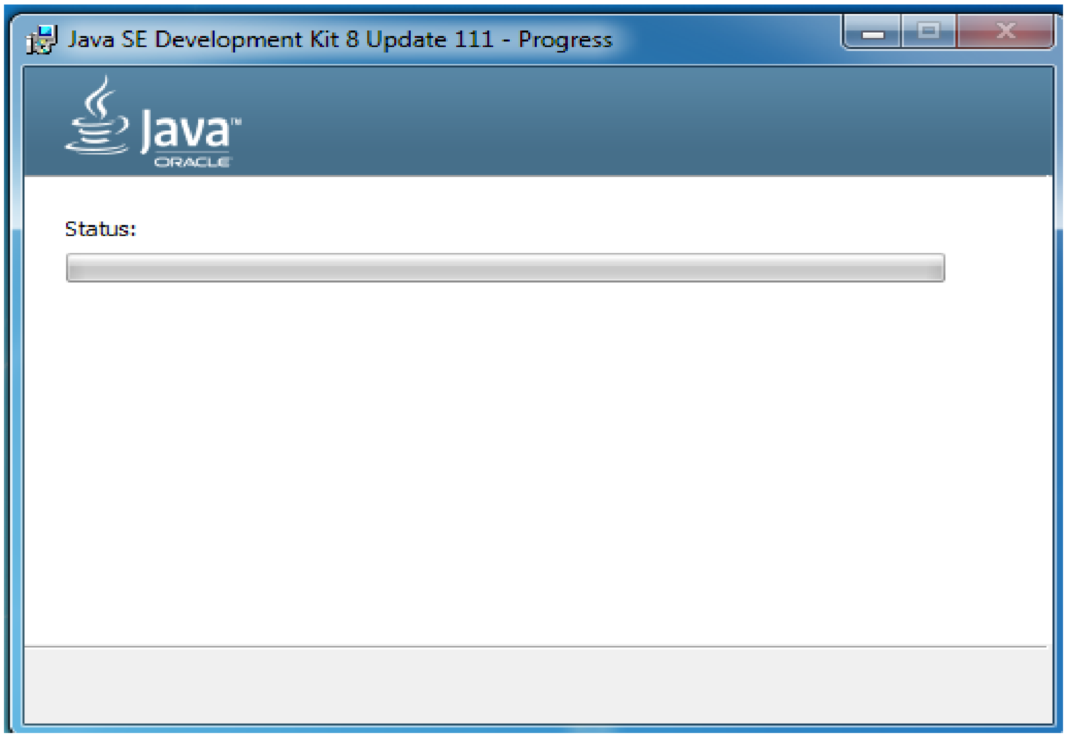

Java SE Development Kit 8 Update 111 - Custom Setup

Select optional features to install from the list below. You can change your choice of features after installation by using the Add/Remove Programs utility in the Control Panel

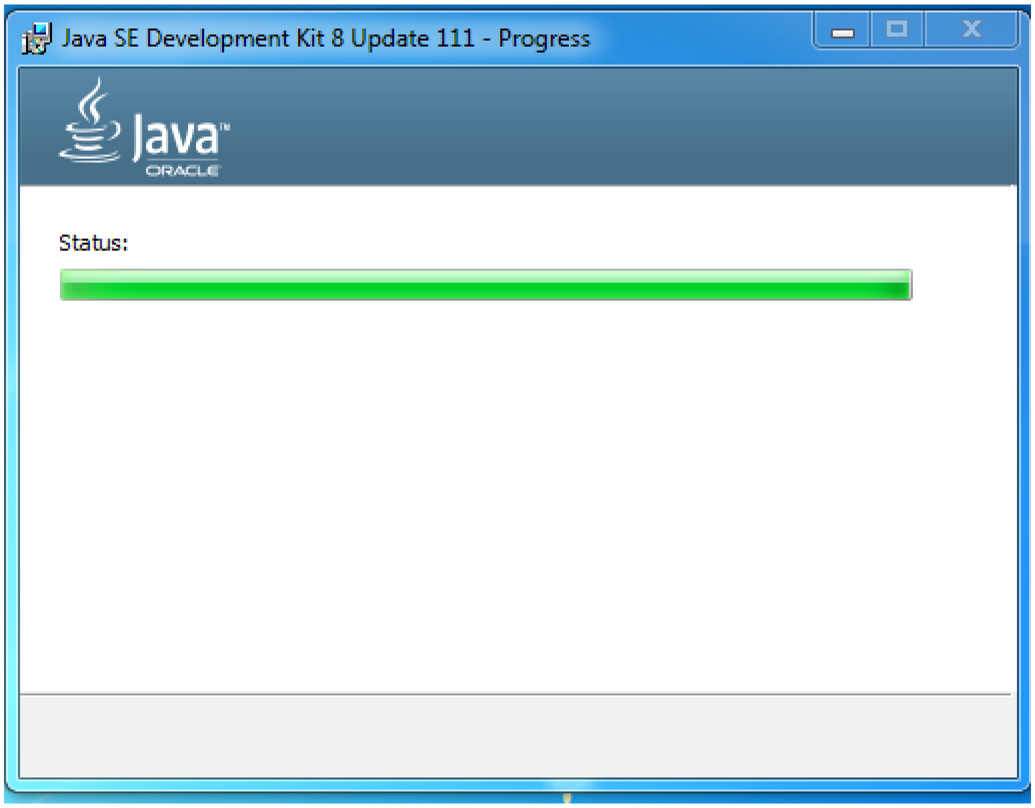
Development Tools
Source Code
Public JRE

Feature Description
Java SE Development Kit 8 Update 111, including the JavaFX SDK, a private JRE, and the Java Mission Control tools suite. This will require 180MB on your hard drive.

Install to:
C:\Program Files\Java\jdk1.8.0_111\

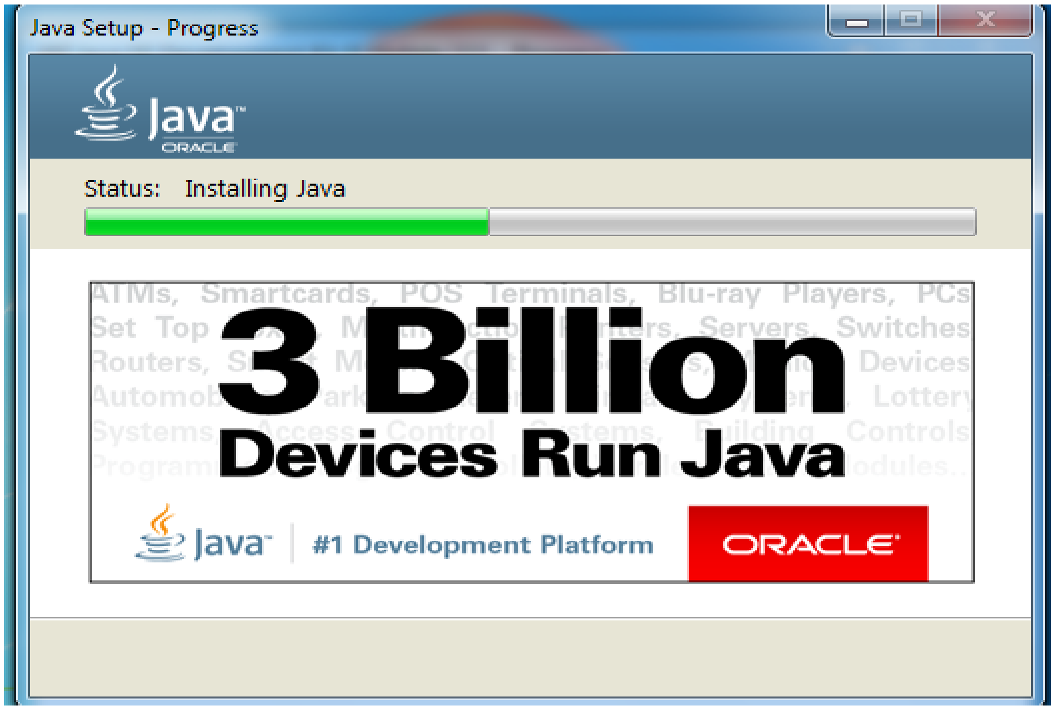

Java SE Development Kit 8 Update 111 - Progress

Status:



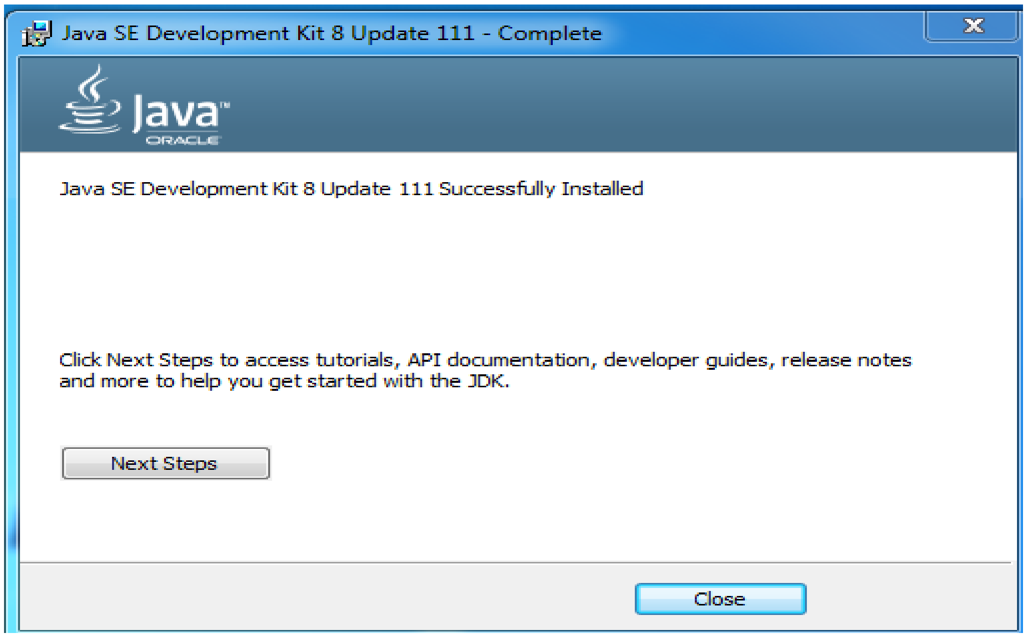
Java SE Development Kit 8 Update 111 - Progress

Status:



Java Setup - Progress

Status: Installing Java


Java SE Development Kit 8 Update 111 - Complete

Java SE Development Kit 8 Update 111 Successfully Installed

Click Next Steps to access tutorials, API documentation, developer guides, release notes and more to help you get started with the JDK.

Finally Click on Close. Now you have successfully installed Java SE software. To use Java SE 8, you have to configure the environment as guided below in detail.

1.7 Configuring JAVA Environment:

As by just installing java standard edition software, we can't run java programs directly. We need to configure the environment first to use java.

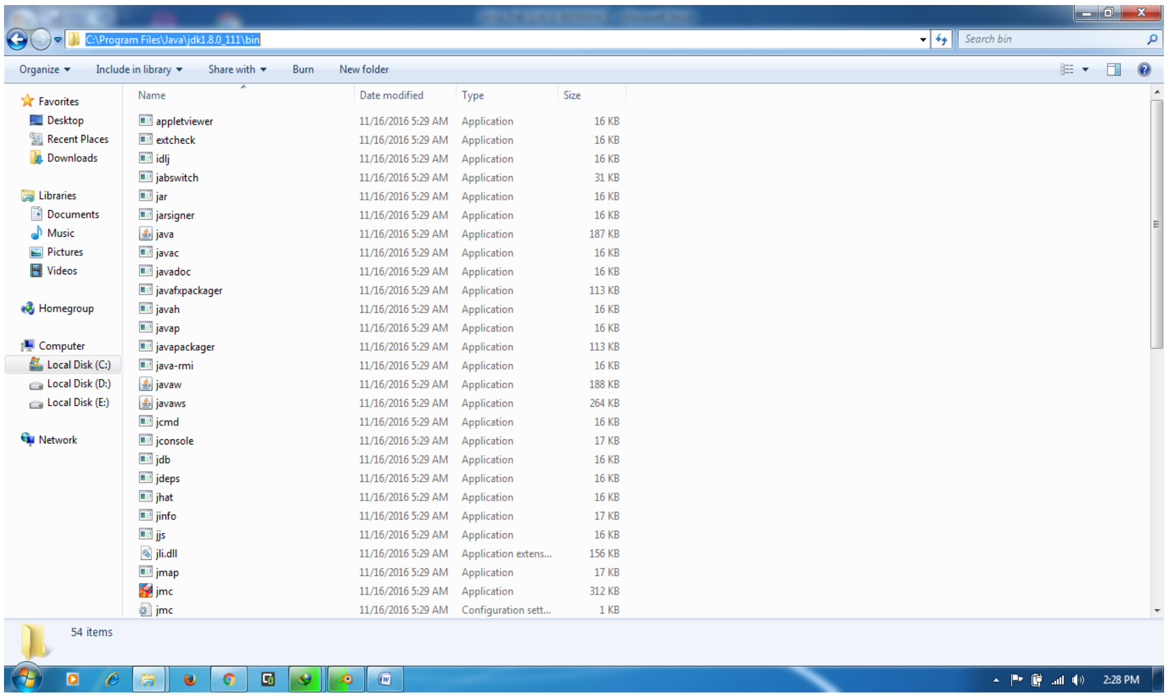
The Environment can be configured by opening system properties and then step by step screen shots as follows:

Step1: First we need to copy the path where the java is installed. Most of the computers by default in Local Disk C drive, in program files we find a folder named java. In that open jdk 8 then open bin folder. Now you need to copy that entire path which can be shown at the top of the window dialogue box.

The path can be as follows:

Path:

C:\Program Files\Java\jdk1.8.0_111\bin

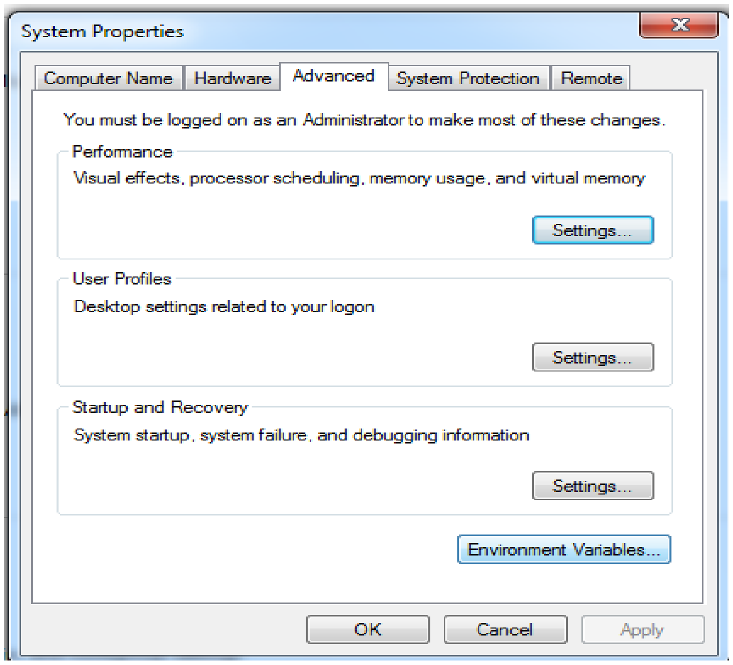


C:\Program Files\Java\jdk1.8.0_111\bin

Name	Date modified	Type	Size
appletviewer	11/16/2016 5:29 AM	Application	16 KB
extcheck	11/16/2016 5:29 AM	Application	16 KB
idlj	11/16/2016 5:29 AM	Application	16 KB
jabswitch	11/16/2016 5:29 AM	Application	31 KB
jar	11/16/2016 5:29 AM	Application	16 KB
jarsigner	11/16/2016 5:29 AM	Application	16 KB
java	11/16/2016 5:29 AM	Application	187 KB
javac	11/16/2016 5:29 AM	Application	16 KB
javadoc	11/16/2016 5:29 AM	Application	16 KB
javafxpackager	11/16/2016 5:29 AM	Application	113 KB
javah	11/16/2016 5:29 AM	Application	16 KB
javap	11/16/2016 5:29 AM	Application	16 KB
javapackager	11/16/2016 5:29 AM	Application	113 KB
java-rmi	11/16/2016 5:29 AM	Application	16 KB
javaw	11/16/2016 5:29 AM	Application	188 KB
javaws	11/16/2016 5:29 AM	Application	264 KB
jcmd	11/16/2016 5:29 AM	Application	16 KB
jconsole	11/16/2016 5:29 AM	Application	17 KB
jdb	11/16/2016 5:29 AM	Application	16 KB
jdeps	11/16/2016 5:29 AM	Application	16 KB
jhat	11/16/2016 5:29 AM	Application	16 KB
jinfo	11/16/2016 5:29 AM	Application	17 KB
jjs	11/16/2016 5:29 AM	Application	16 KB
jli.dll	11/16/2016 5:29 AM	Application extens...	156 KB
jmap	11/16/2016 5:29 AM	Application	17 KB
jmc	11/16/2016 5:29 AM	Application	312 KB
jmc	11/16/2016 5:29 AM	Configuration sett...	1 KB

54 items

Step 2: Now open My Computer properties, there you can find Advanced System Settings especially in windows, then by clicking on that you may notice, Environment Variables option.


System Properties

Computer Name | Hardware | Advanced | System Protection | Remote

You must be logged on as an Administrator to make most of these changes.

Performance
Visual effects, processor scheduling, memory usage, and virtual memory
Settings...

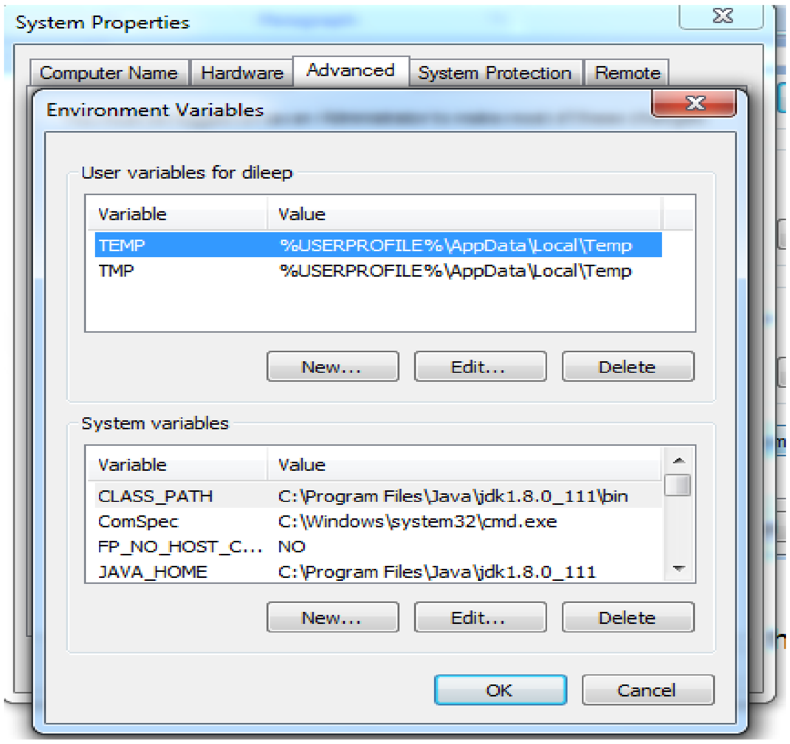
User Profiles
Desktop settings related to your logon
Settings...

Startup and Recovery
System startup, system failure, and debugging information
Settings...

Environment Variables...

OK | Cancel | Apply

Step 3: Now Click on Environment variables option, then the following window opens:


Environment Variables

User variables for dileep

Variable	Value
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

New... | Edit... | Delete

System variables

Variable	Value
CLASS_PATH	C:\Program Files\Java\jdk1.8.0_111\bin
ComSpec	C:\Windows\system32\cmd.exe
FP_NO_HOST_C...	NO
JAVA_HOME	C:\Program Files\Java\jdk1.8.0_111

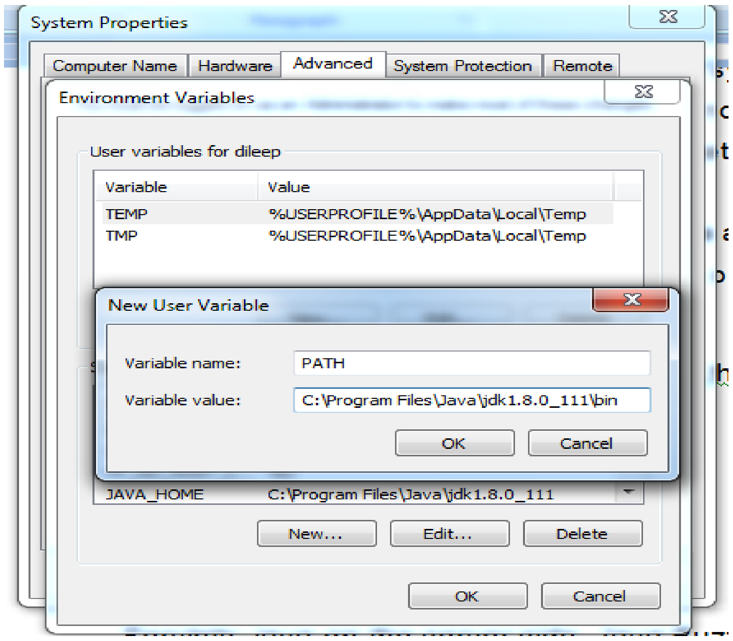
New... | Edit... | Delete

OK | Cancel

Environment variables are a set of dynamic named values that can affect the way running processes will behave on a computer. They are part of the **environment** in which a process runs. Here you will see two types of variables that are system variables and user variables. The important point to be noted is you must not change any previous variables. If anything gets changed then your system will not function properly.

Now you can set java path in any one among that variable. Preferably set java path on user variables or else you can set in both.

Step 4: Click on New, in that dialogue box..Then will be displayed as follows: Also you can type as it is Path as Name and Value as that is already you have copied.



Variable	Value
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

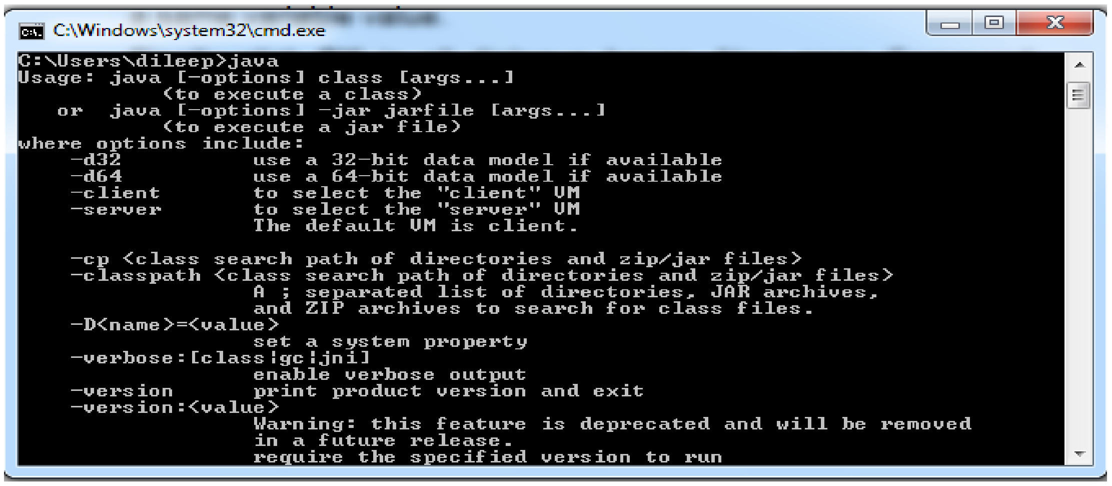
Variable name: PATH

Variable value: C:\Program Files\Java\jdk1.8.0_111\bin

JAVA_HOME C:\Program Files\Java\jdk1.8.0_111

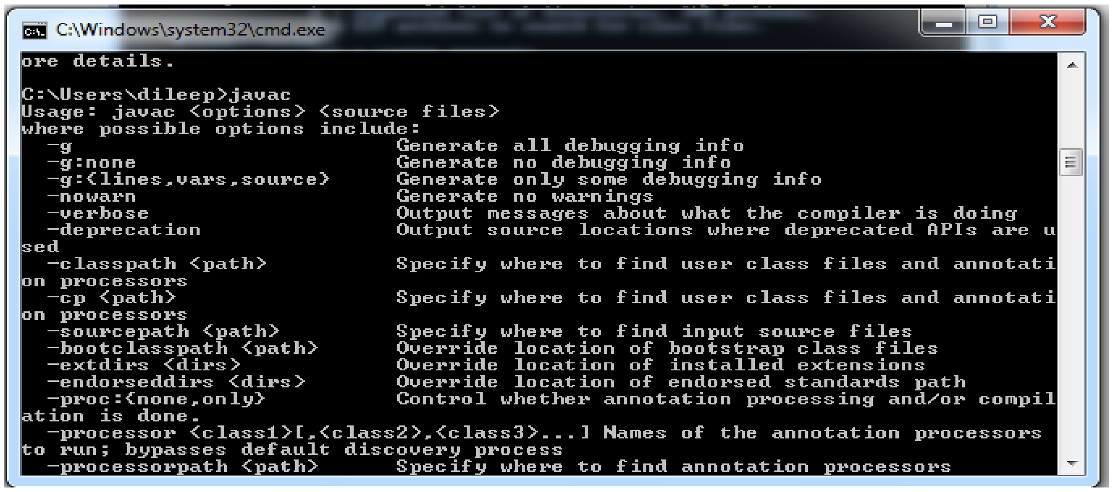
In the same now again click new and set CLASS_PATH also with a same variable value.

Finally click **OK** in all dialogue boxes. Now open Command Prompt and type **java,** then it must display as follows if not you didn't set path correct.



```
C:\Users\dileep>java
Usage: java [-options] class [args...]
           (to execute a class)
   or  java [-options] -jar jarfile [args...]
           (to execute a jar file)
where options include:
    -d32          use a 32-bit data model if available
    -d64          use a 64-bit data model if available
    -client       to select the "client" VM
    -server       to select the "server" VM
                  The default VM is client.

    -cp <class search path of directories and zip/jar files>
    -classpath <class search path of directories and zip/jar files>
                  A ; separated list of directories, JAR archives,
                  and ZIP archives to search for class files.
    -D<name>=<value>
                  set a system property
    -verbose:[class|gc|jni]
                  enable verbose output
    -version      print product version and exit
    -version:<value>
                  Warning: this feature is deprecated and will be removed
                  in a future release.
                  require the specified version to run
```

And also in a similar way type **javac** will display in command prompt as follows:



```
ore details.

C:\Users\dileep>javac
Usage: javac <options> <source files>
where possible options include:
  -g                         Generate all debugging info
  -g:none                    Generate no debugging info
  -g:{lines,vars,source}     Generate only some debugging info
  -nowarn                    Generate no warnings
  -verbose                   Output messages about what the compiler is doing
  -deprecation               Output source locations where deprecated APIs are u
sed
  -classpath <path>          Specify where to find user class files and annotati
on processors
  -cp <path>                 Specify where to find user class files and annotati
on processors
  -sourcepath <path>         Specify where to find input source files
  -bootclasspath <path>      Override location of bootstrap class files
  -extdirs <dirs>            Override location of installed extensions
  -endorseddirs <dirs>       Override location of endorsed standards path
  -proc:{none,only}          Control whether annotation processing and/or compil
ation is done.
  -processor <class1>[,<class2>,<class3>...] Names of the annotation processors
to run; bypasses default discovery process
  -processorpath <path>      Specify where to find annotation processors
```

Now the Java Environment is successfully configured in your computing system. The following are some of the commands used in java.

- javac - The Java Compiler
- java - The Java Interpreter
- jdb- The Java Debugger
- appletviewer -Tool to run the applets
- javap - to print the Java bytecodes
- javaprof - Java profiler
- javadoc - documentation generator
- javah - creates C header files

1.8 Servlets: java on the server side:

In the coming days more number of servers support server-side Java, including the Netscape FastTrack 2.0 and Enterprise 2.0 servers, JavaSoft's Java Web Server (formerly called Jeeves), the World Wide Web Consortium's Jigsaw, WebLogic's T3Server, Oracle's Webserver, and Peak Technologies' ExpressO. Each of these servers uses a different server-side Java API, requiring developers to write different programs for each server they will be using.

Server-side Java (SSJ), sometimes called servlets or server-side applets, is a powerful hybrid of the Common Gateway Interface (CGI) and lower-level server API programming -- such as NSAPI from Netscape and ISAPI from Microsoft.

Some of the advantages are as follows:

- Run compiled code across platforms as we now java is a platform independent.
- Memory Management (Garbage Collection).
- Large developer market.
- Easy migration for C++ developers that is any c++ developer can easily change to a java developer by learning some more concepts.
- Java is more secure

1.9 Java Buzzwords:

Java Buzzwords are also called as Java features. The Developers who developed java language wants to solve all the issues which cannot solve through modern languages.

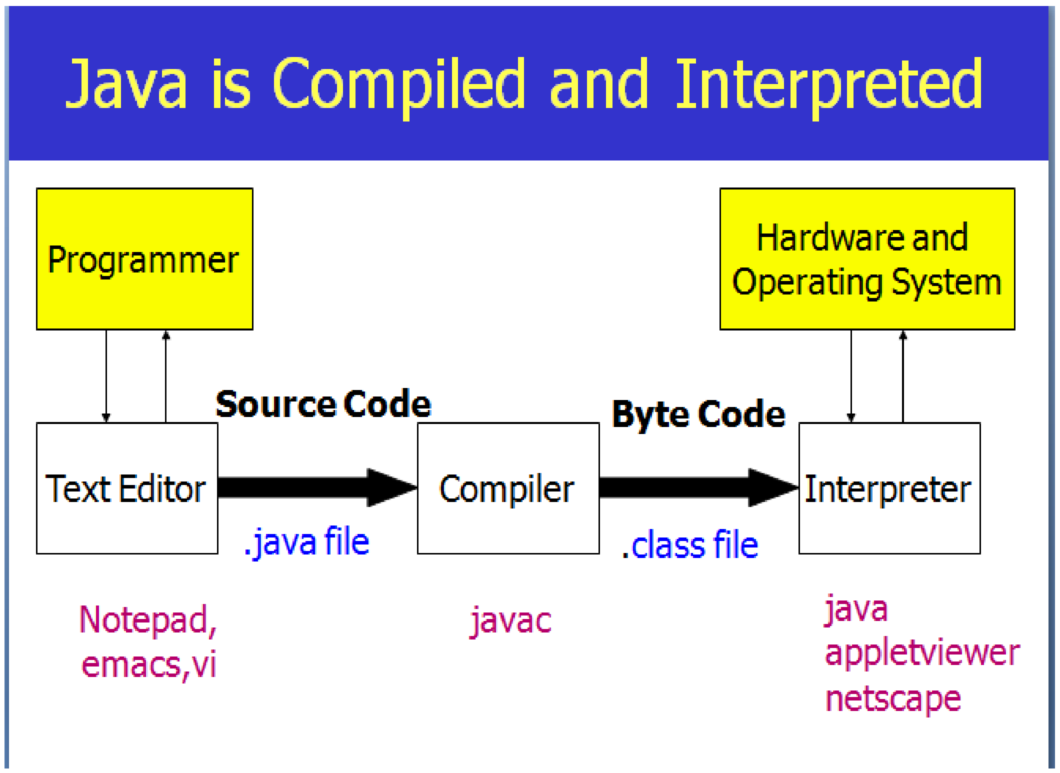
1. Object Oriented.
2. Compiled and interpreted.
3. Platform independent and portable.
4. Distributed.
5. Robust and Secure.
6. Familiar, Simple and Small.
7. Multithreaded and interactive.
8. Dynamic and Extensible.
9. High Performance.

1. **Object Oriented:**

   Everything in java is described in the form of an object .complete program code and data resides in object and class.

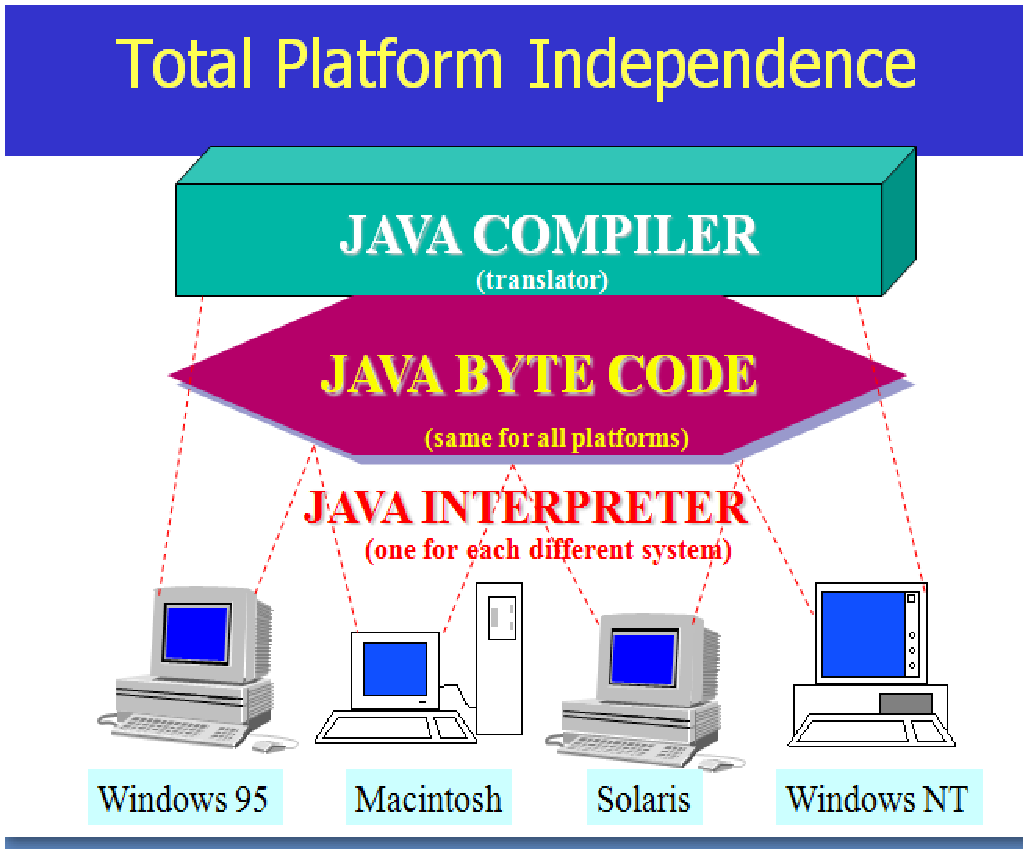
2. **Compiled and interpreted:**

   Generally a Program should be either Compiled or interpreted but java combines both these approaches. Initially java program will be compiled which generates Byte Code using JVM (java Virtual Machine) which Is not machine instruction, So java interpreter is used to convert Byte Code to Machine code and executes the java program. In this way java is not only compiled but also interpreted.



3. Platform independent and Portable:

Java program can be executed on any operating system and on any machine which we can say it is a platform independent language. And it is portable because it can be moved from one system to another.



4. Distributed:

Java applications can open and access remote objects on internet very easily. This enables many programmers residing at different locations to work on a single project.

5. Robust & Secure:

Java has strict compile time and run time checking for data types and it also incorporates the concept of exception handling, which captures errors and eliminates the risk of system crash. Because of these reasons Java is said to be Robust.

Java is more secured as compared to any other programming language. Java not only verifies the memory access but also ensures that no virus communicates with an applet.

6. **Multithreaded & Interactive:**

   Java can handle several tasks simultaneously which is called multithreading. The Java runtime comes with tools which supports multiprocessor synchronization and construct interactive systems running smoothly.

7. **Dynamic & Extensible:**

   Java is capable of dynamically linking in new class libraries, Methods and objects. Java also supports extensibility. It supports functions written in other languages like C and C++.

8. **High Performance:**

   Java using the concept of Byte Code which makes it works more powerful and as the performance increases.

1.9.1 COMPARISON OF JAVA, C AND C++:

JAVA & C++:

JAVA	**C++**
Java is Object oriented programming language.	Partially object oriented programming.
Java is more secure because It doesn't use pointers.	It is not Secure. It uses Pointers
Java follows bottom up approach.	It follows top down approach or both.
It doesn't support multiple inheritance directly. In order to use this we need interfaces concept.	It supports all types of inheritance concepts
Java byte code is platform independent.	C++ is platform dependent.
Java support automatic garbage collection for memory cleaning automatically.	C++ support destructors, which is automatically invoked when the object is destroyed.
Java has built in support for threads.	C++ has no built in support for threads.
Java has method overloading, but no operator overloading.	C++ supports both method overloading and operator overloading.
There is no *goto* statement.	C++ has *goto* statement.

1.9.2 JAVA & C: The following are the some of the major differences between Java and C programming languages.

C (Procedure oriented)	**JAVA (Object oriented)**
C doesn’t support object oriented programming concepts	Java support object oriented programming concepts
C follows top down approach	Java follows bottom up approach
C is platform dependent	Java is platform independent.
C is less secure.	Java is more secure because It doesn’t use pointers.
It uses header files	It uses packages
It supports pointers	It won’t use pointers
It doesn’t support inheritance	It uses inheritance concepts
It uses static memory allocation	It uses dynamic memory allocation

CHAPTER 2
OBJECT ORIENTED THINKING

2.0 Introduction to Object Oriented Programming (OOP):

Object Oriented Programming is the process of modularizing program by creating partitioned memory area for both data and methods that can be used as templates for creating such copies of modules on demand.

OOP allow dividing a big problem into small problems of entities called objects and then builds data and methods around those entities. Java is pure object oriented programming when compared to c++ but Ada is completely object oriented programming because it doesn’t uses primitive data types.

The principles of Object Oriented Programming are:

- Class
- Object
- Encapsulation
- Inheritance
- Polymorphism
- Data Abstraction
- Dynamic Binding

Class:

Class is a blueprint of containing attributes and methods.

Or

Class is a collection of objects of similar type.

Syntax:

```
        class ClassName {
        // variables or attributes
        // methods
        }
```

For example:

```
class Room{
int length;
int width;  // attributes
int height;
void volumeOfRoom(){
//method definition
}
}
```

Object:

Object is a basic runtime entity or Object can be an instance of class.

An object can be created in the following ways:

By statically or reference

ClassName objname;

By dynamically an object can be created using **new** keyword.

ClassName objname=new ClassName();

Encapsulation:

Encapsulation is the process of wrapping up off data and methods into a single unit. Here the data is not accessed by outside the world. This is how data hiding is possible in object oriented programming.

Inheritance:

Inheritance is the process of acquiring the properties of a base class object to the properties of the derived class object.

Java doesn’t support multiple inheritance but to use this the concept of interfaces should be used.

This concept provides the idea of CODE REUSABILITY in OOPS.

For acquiring properties from one class to another class **extends** keyword is used.

Syntax:

```
class BaseClass{
//attributes and methods
}
class   Derived extends BaseClass
{
// attributes and methods
}
```

Polymorphism:

Polymorphism is the phenomenon of representing more than one form. This concept aims to the use of FUNCTION OVERLOADING.

Data Abstraction:

It is the process of representing essential features without including the background details or any explanation.

Dynamic Binding:

It refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding is also called as RUN-TIME BINDING.

Need for OOP paradigm:

Object Oriented Programming provides awesome features in programming like code reusability and extensibility, modularity, easier to maintain and design stability. These are some of the reasons for using OOP concepts in JAVA programming.

Object-oriented-programming is a bottom-up approach which eliminates some of the disadvantages of structured programming language. The central concept of object-oriented programming is the object, which consists of data and subroutines. An object is a kind of entity that has an internal state and that can respond to Messages. The data of an object can be accessed only by the methods associated with the object.

Some of the Object-Oriented Paradigm are:

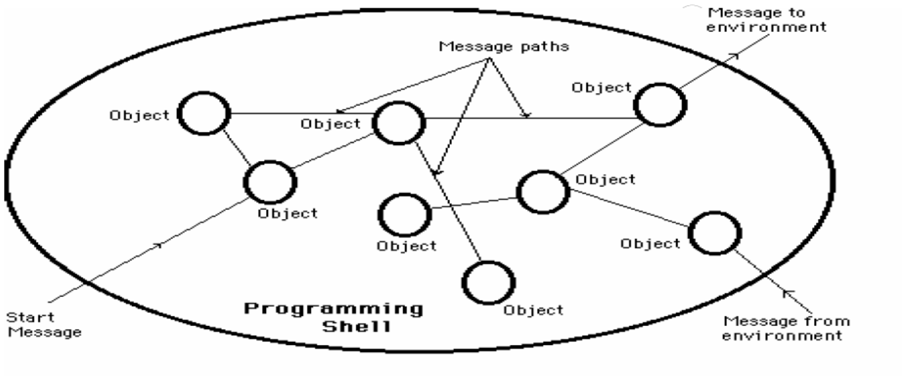
- Emphasis is on data rather than procedure.

- Programs are divided into objects.
- Data Structures are designed such that they Characterize the objects.
- Methods that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through methods.

2.1 A way of viewing world:

Agents:

An agent is a real-time object.



Responsibility:

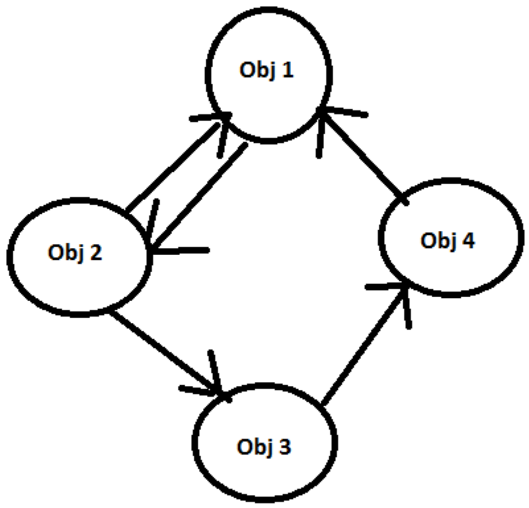
Object oriented programming is platform independent due to byte code. Classes creates objects

Messages:

The communication between two objects is called messages. Steps for message passing are

- Creating class
- Creating objects

➢ Communication between objects



Syntax for message passing is objectname.methodName();

Methods:

Method is defined as group of statements which perform specific operations.

Syntax:

```
Returntype methodName(arguments){
//body or definition of the method
}
```

Classes and Instances:

Class is a collection of objects and object is the instance of a class.

2.2 Method Binding:

Method Binding refers to the linking of a procedure call to the code to be executed in response to the call. Method binding occurs at run time.

For example,

```
Import java.io.*;
class method
{
void test()
{
System.out.println(“ Methods have no arguments”);
}
void test(int a)
{
System.out.println(“ Methods have 1 arguments”);
}
void test(int a, int b)
{
System.out.println(“a value is “+a+”b value is”+b);
}
void test(int a)
{
return a*a;
}
class Mbinding
{
public static void main(String args[])
{
int x;
method obj=new method(); //creating an object
obj.test();//message passing
obj.test(5);
x=obj.test(5);
System.out.println("The x value is "+c);
}
}
```

2.3 Overriding and Exceptions:

Overriding means a method in a derived class which is already defined in base class.

For example,

```
Import java.io.*;
class Super{
void callMe(){
System.out.printl("Inside super class call me");
}
}
class   Sub extends Super
{
void callMe(){
System.out.printl("Inside sub class call me");
}
}
class MethodOverriding
{
```

```
public static void main(String args[])
{
Sub obj=new Sub();
Obj.callMe();
}}
```

2.4 Exceptions:

An event which occurs during the execution of program is called exception. These exceptions are of two types. They are compile time and run time exceptions.

When any unavoidable circumstances occur in our program then exceptions are raised by handling control to specific functions called handlers. We use some keywords to handle some exceptions. They are

- Try
- Catch
- Throw
- Throws
- Throwables

2.5 Coping with complexity:

If we take small programs, less amount of time to execute. And if we take large program, requires large amount of time for execution.

2.6 Abstraction Mechanism:

It is the act of representing essential features without including background details or explanation.

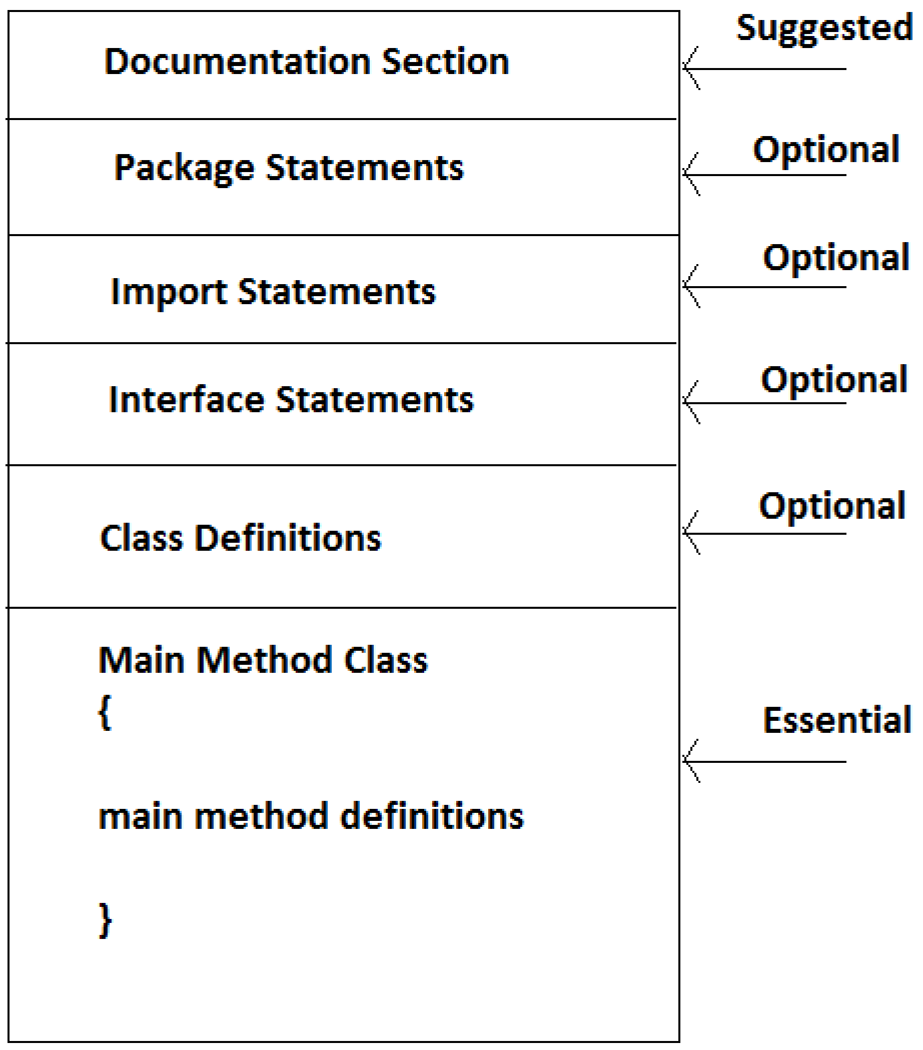
To make abstract class, we use **abstract** keyword.

For example,

```
abstract class ClassName{
//attributes
void method();// abstract class contains only method signatures
}
```

2.7 General Structure of a JAVA program:

A Java program may contain many classes of which only one class defines a main method. Class contains data members and methods that operate on the data member of the class. To write a JAVA program we first define classes and then put together. The structure of a java program is somewhat different from c and c++. In java we use packages and import statements whereas in c & c++ header files are used. The differences between the programming languages are given in Chapter 1. A java program may contain one or more sections as shown below:



- ➢ Documentation section contains comments which are most useful for the purpose of understanding.
- ➢ Packages are the collection of classes and methods. Packages may be user defined.
- ➢ Import statements contain predefined classes and methods.

- Interface is a collection of data members and method signatures. And these can be implemented in other classes.
- A JAVA program must contains at least a class definition including main method in it. Also a JAVA program can be written without using a main method and a class definition.
- This can be done by using a static method. Because compiler or interpreter invokes first static methods. These are invoked without the help of an object creation.
- Finally a JAVA program can be saved as a main Class name with the extension of .java (dot java).

A Simple JAVA Program:

```
Class TestExample
{
public static void main(String args[])
{
      System.out.println(“ Welcome Jayaprakash Java Quick Ref. “);
}
}
```

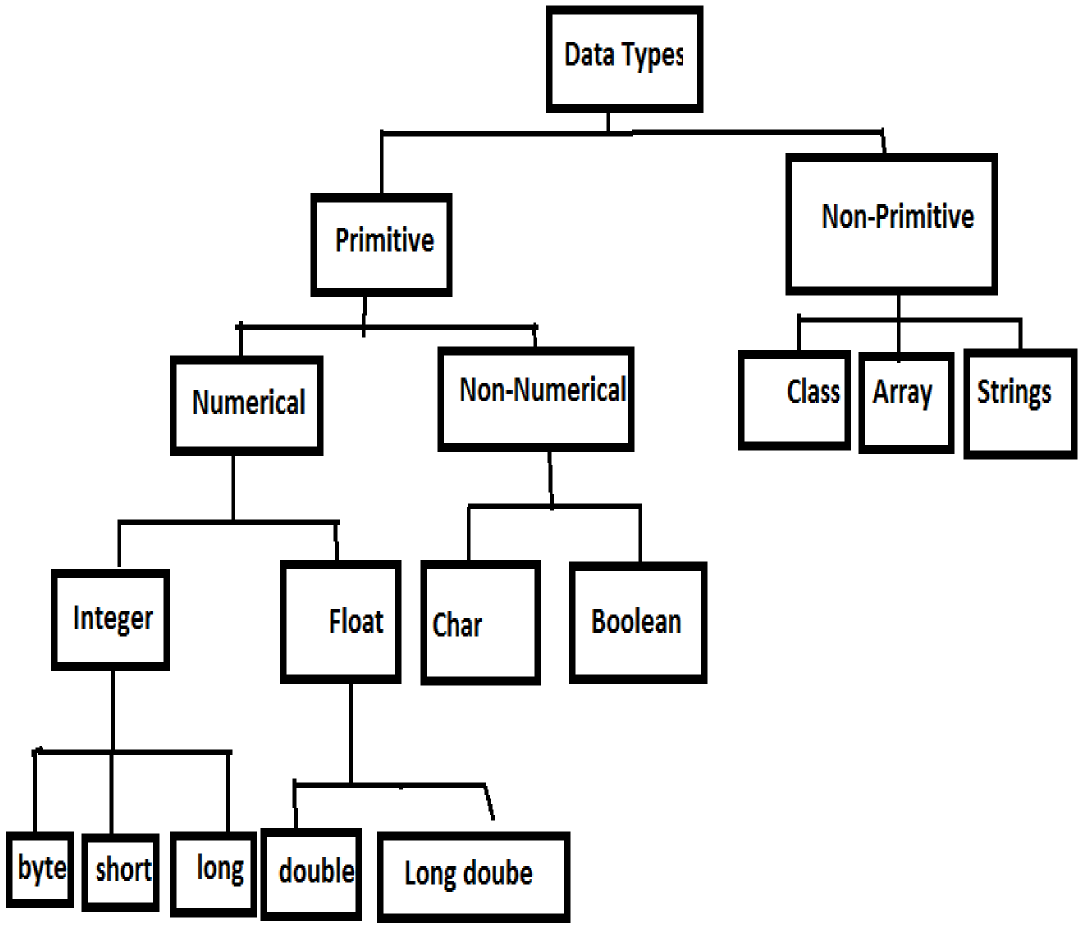
This program has to be saved as **TestExample.java**

CHAPTER 3
JAVA BASIC CONCEPTS OF PROGRAMMING

3.0 Data Types:

The type of value stored in a variable which is a named memory location called Data Type.

The classification of data types as given below in a diagrammatic way:



3.1 Primitive Data Types:

Integer:

Data Type	Size in Bytes	Bits	Range
Byte	1	8	-128 to 127
Short	2	16	-32768 to 32767
Int	4	32	-2^31 to 2^31-1
Long	8	64	-2^63 to 2^63-1

Float:

Data Type	Size in Bytes	Bits	Range
Float	4	32	4.9e^-324 to 1.8e^38
Double	8	64	1.4e-^-45 to 3.4e^38

Character:

Data Type	Size in Bytes	Bits	Range
Char	2	16	0 to 65536

Boolean:

Data Type	Size in Bytes	Bits	Range
Boolean	1	8	T/F, 0 or 1

Syntax for using data types as follows:

Datatype variablename;

3.2 Variables:

A variable is a named location which stores a value in it.

Rules for naming a variable are as follows:

- A variable should not start with a number
- Variable uses only a single special character called underscore (_).
- Variable may contain one or more words but uses _ symbol.

Syntax: datatype variablename;

Or

datatype var1,var2…varn;

For example: int a;

Variables can be initialized in two ways;

Static initialization can be as int c=5;

Dynamic initialization can be as follows:

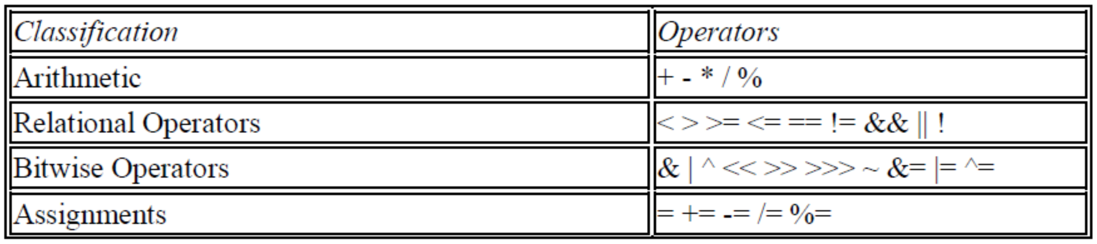
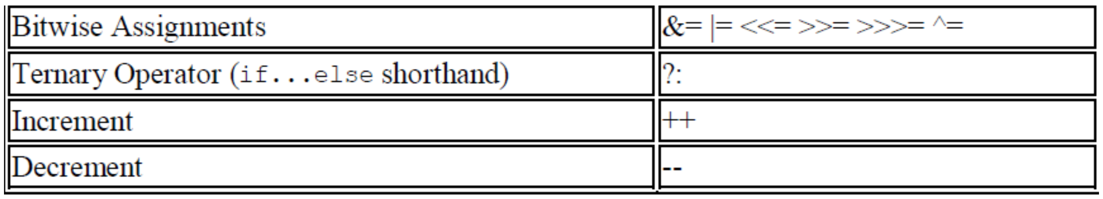
Variablename=expression

C=a^2+b^2

In Java only local variables are declared. No global variables are declared.

Operators:

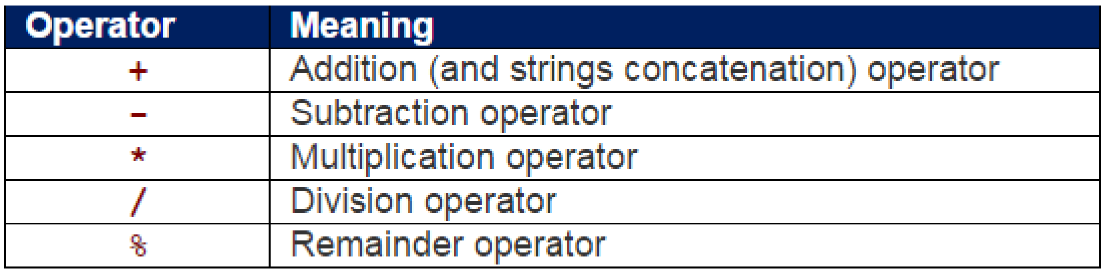
An operator is a symbol which performs specific operation. Operators are of various types. Java has strict definitions of operators. It doesn't allow for overloading, Java operators are used to build value expressions. Java provides two types of operators: binary and unary. Binary operators are used to compare two values. Unary operators use a single value. All of Java's binary and unary operators can be found in below table

Classification	*Operators*
Arithmetic	+ - * / %
Relational Operators	< > >= <= == != && || !
Bitwise Operators	& | ^ << >> >>> ~ &= |= ^=
Assignments	= += -= /= %=
Bitwise Assignments	&= |= <<= >>= >>>= ^=
Ternary Operator (`if...else` shorthand)	?:
Increment	++
Decrement	--

They are:

- **Arithmetic Operators**



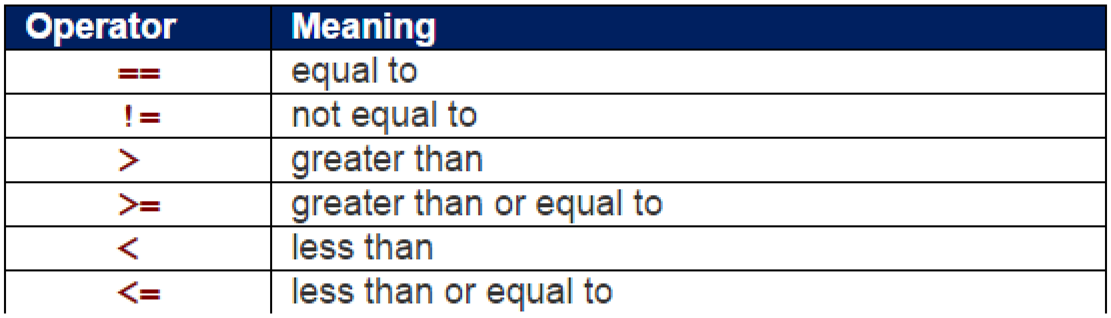
Operator	Meaning
+	Addition (and strings concatenation) operator
-	Subtraction operator
*	Multiplication operator
/	Division operator
%	Remainder operator

An Example program which illustrate arithmetic operators:

```
public class ArithmeticExample {
   public static void main(String[] args) {
      int x = 40;
      int y = 20;
      int r= x + y;
      System.out.println("x + y = " + r);
      r = x - y;
      System.out.println("x - y = " + r);
```

```
        r = x * y;
        System.out.println("x * y = " + r);
        r = y / x;
        System.out.println("y / x = " + r);
        r= x % 3;
        System.out.println("x % 3 = " + r);
    }
}
```

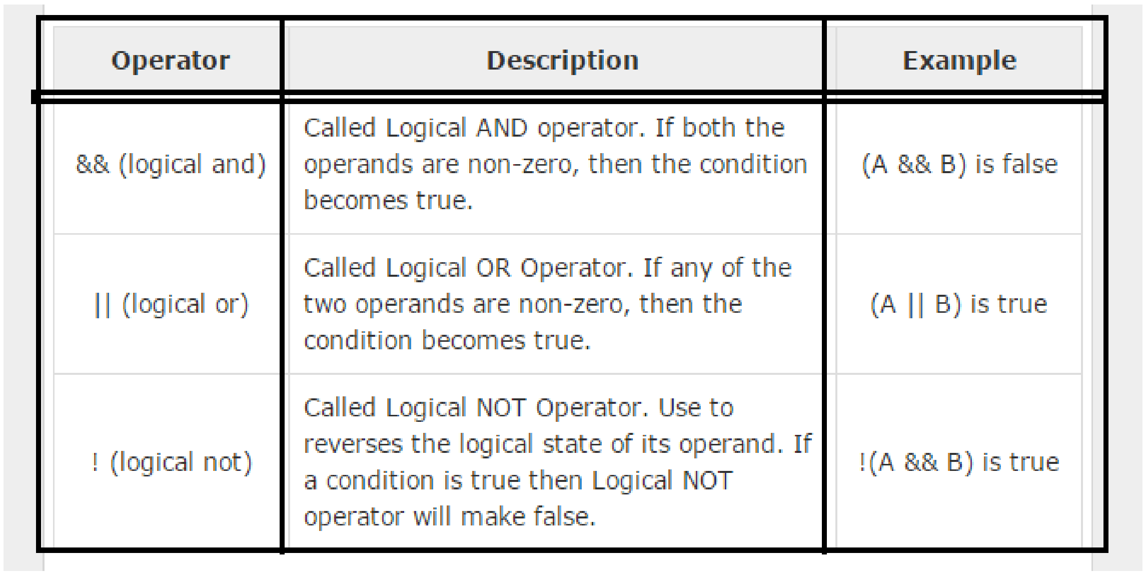
➢ **Relational Operators**



Operator	Meaning
==	equal to
!=	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

```
        public class RelationalDemo {
        public static void main(String[] args) {
        int x = 10;
        int y = 20;
        boolean result = x == y;
        System.out.println("x == y? " + result);
        result = x != y;
        System.out.println("x != y? " + result);
        result = x > y;
        System.out.println("x > y? " + result);
        result = x >= y;
        System.out.println("x >= y? " + result);
        result = x < y;
        System.out.println("x < y? " + result);
        result = x <= y;
        System.out.println("x <= y? " + result);
    }
}
```

- **Logical Operators**



Operator	Description	Example
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false
|| (logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true
! (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true

```
class LogicalOperatorDemo
{
public static voidmain(String args[ ])
{
boolean i = Boolean.valueOf(args[0]).booleanValue();
boolean j = Boolean.valueOf(args[1]).booleanValue();
System.out.println("i = " + i);
System.out.println("j = " + j);
System.out.println("Logical operators:");
System.out.println("i & j " + (i & j));
System.out.println("i | j " + (i| j));
System.out.println("i ^ j " + (i^ j));
System.out.println("!i " +!i);
System.out.println("i && j " + (i && j));
System.out.println("i || j " + (i || j));
System.out.println("i == j " + (i == j));
System.out.println("i != j " + (i !=j));
}}
```

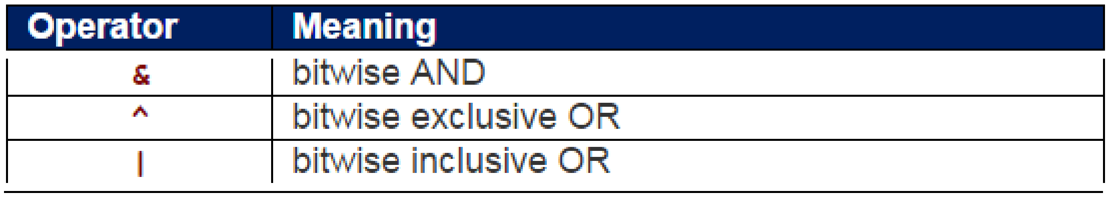
- Increment/Decrement operators

```
public class PrefixPostfixDemo {
    public static void main(String[] args) {
        int x = 10;
        int y = 20;
        System.out.println(++x);
        System.out.println(x++);
        System.out.println(x);
        System.out.println(--y);
        System.out.println(y--);
        System.out.println(y);
    }}
```

Output:

```
11
11
12
19
19
18
```

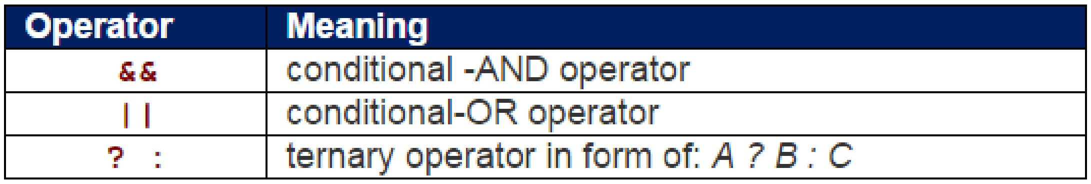
- **Bitwise Operators**



Operator	Meaning
&	bitwise AND
^	bitwise exclusive OR
|	bitwise inclusive OR

```
class BitwiseOperators
{
public static voidmain(String args[ ])
{
char c = 'A';
byte b = 100;
short s = 100;
int i = -100;
long lo = 100;
System.out.println(c & 0xf);
System.out.println(b | 1);
System.out.println(s ^ 1);
System.out.println(~i);
System.out.println(lo | 1);
System.out.println(i >> 2);
System.out.println(s >>> 2);
System.out.println(i << 2);
}}
```

- **Conditional Operators**



Operator	Meaning
&&	conditional -AND operator
||	conditional-OR operator
? :	ternary operator in form of: *A ? B : C*

```
class TernaryOperatorDemo
{
public static voidmain(String args[ ])
{
int i = 10;
int j = 5;
System.out.println((i > j) ? i : j);
}}
```

- **Special Operator**

Special operator called dot operator is used in java. This operator is used for message passing.

For example;

```
 objname.methodname();
```

3.3 PROGRAM CONTROL STATEMENTS:

Selection/Decision statements:

Selection Statements allow the program to choose different parts of the execution based on the outcome of an expression.

(Or)

Selection Statements allow to control the flow of program's execution based upon conditions known only during run-time. Java provides four selection statements:

1) if

```
class IfDemo
{
public static voidmain(String args[ ])
{
if(args.length ==0)
System.out.println("You must have command line arguments");
}}
```

OUTPUT :

COMPILE :

```
javac IfDemo.java
```

EXECUTION :

```
java IfDemo
    You must have command line arguments
```

2) if-else

```
class SquareRoot
{
public static voidmain(String args[ ])
{
double d = Double.valueOf(args[0]);
if(d < 0)
System.out.print(Math.sqrt(-d) + "i");
else
System.out.print(Math.sqrt(d));
} }
```

OUTPUT :

```
COMPILE :
javac SquareRoot.java
EXECUTION :
java SquareRoot 100000.000000
316.22776601683796
```

3) if-else-if

```
class IfElse
{
public static voidmain(String args[ ])
{
int i = Integer.parseInt(args[0]);
if(i > 0)
if(i > 3)
System.out.println("i > 0 and i > 3");
else
System.out.println("i > 0 and i <= 3");
else
System.out.println("i <= 0");
}}
```

OUTPUT :

```
COMPILE :
javac IfElse.java
EXECUTION :
java IfElse 2
i > 0 and i <= 3
```

4) **switch**

```
class SwitchDemo
{
public static voidmain(String args[ ])
{
int i = Integer.parseInt(args[0]);
switch(i)
{
case 1:
```

```
System.out.println("one");
break;
case 2:
System.out.println("two");
break;
case 3:
System.out.println("three");
break;
case 4:
System.out.println("four");
break;
default:
System.out.println("Unrecognized Number");
} } }
```

OUTPUT:

```
COMPILE:
javac SwitchDemo.java
EXECUTION :
java SwitchDemo 3
three
```

Iteration/loop control statements:

Iteration Statements enable repeated execution of part of a program until a certain termination condition becomes true. Java provides three iteration statements.

1) WHILE

```
class WhileDemo
{
public static voidmain(String args[ ])
{
int i = Integer.parseInt(args[0]);
while(i > 0) {
System.out.print(i + " ");
i--;
} } }
```

OUTPUT :

COMPILE :

```
javac WhileDemo.java
```

EXECUTION :

```
java WhileDemo 6
    6 5 4 3 2 1
```

2) DO-WHILE

```
class Fibonacci
{
public static voidmain(String args[ ])
{
int count = 0;
int i = 0;
int j = 1;
do {
```

```
System.out.print(j + " ");
int k = i + j;
i = j;
j = k;
} while(++count < 15);
}}
```

OUTPUT :

COMPILE :

```
javac Fibonacci.javaa
```

EXECUTION :

```
java Fibonacci
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

3) FOR

```
public class ForDemo
{
public static voidmain(String args[ ])
{
for(int num = 1; num < 11; num = num + 1)
System.out.print(num +" ");
System.out.println("terminating");
  } }
```

OUTPUT :

COMPILE :

```
javac ForDemo.java
```

EXECUTION :

```
java ForDemo
1 2 3 4 5 6 7 8 9 10 terminating
```

Jump/control statements:

Jump statements enable your program to execute in a non-linear fashion.

(or)

Jump statements enable transfer of control to other parts of program. Java provides three jump statements:

1) **BREAK**

Break is used to terminate a program from the execution process.

```
    class BreakDemo
    {
public static void main(String args[ ])
{
int i;
for(i = 1; i < 100; i++)
{
System.out.println(i);
if(i == 10) break;
} } }
```

OUTPUT:

COMPILE:

```
javac BreakDemo.java
```

EXECUTION:

```
java BreakDemo
```

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

2) CONTINUE

```
class ContinueDemo
{
public static voidmain(String args[ ])
{
for(int x = 0; x <100; x++)
{
if(x < 95) continue;
System.out.println(x);
} } }
```

OUTPUT :

COMPILE :

```
javac ContinueDemo.java
```

EXECUTION :

```
java ContinueDemo
95
96
97
98
```

3) RETURN

Return is a statement which can be used in any method that which returns a value.

For example return 0;

3.4 ACCESS MODIFIERS/SPECIFIERS:

Variables and methods are collectively known as members. Classes can protect their members by the member access modifiers. There are 3 types of access members in java. They are

- Public: Used for making a class visible for an entire program
- Private: Specified for a class to make it secure.
- Protected: Used under the concept of inheritance.

3.5 The Java keywords - this, super, final and static

THIS KEYWORD:

This Keyword is used as a reference to the object on which the method was invoked. When an object wants to call itself we use this keyword in programming.

An example program using this keyword as follows:

```
class Test
{
```

```
double x;
double y;
double z;
Test(double x, double y, double z)
{
this.x = x;
this.y = y;
this.z = z;
} }
class ThisKeywordDemo
{
public static void main(String args[])
{
Test ob = new Test(1.1, 3.4, -2.8);
System.out.println("value of x : " + ob.x);
System.out.println("value of y : " + ob.y);
System.out.println("value of z : " + ob.z);
}}
```

OUTPUT :

```
Value of x : 1.1
Value of y: 3.4
Value of z: -2.8
```

SUPER KEYWORD:

1. It is used inside a sub-class method body to call a method defined in the parent class. Private methods of the super-class cannot be

called and only public and protected methods can be called by the super keyword only.

2. It is also used in class constructors to invoke constructors of its parent class.
3. **Super** keyword in java also acts as a reference variable that is used to indicate base class object.
4. **Super** is an implicit keyword created in Java Virtual Machine and supply each and every java program for performing important role in three places as shown below:.
   - variable level
   - method level
   - constructor level

Use of super keyword:

Whenever there is derived class, it inherits the base class features, there is a possibility in base class properties that are similar to derived class properties and JVM gets confusion. In order to distinguish between base class features and derived class features must be preceded by super keyword in programming with java.

Syntax: super.baseclass features

Super at variable level:

Whenever the derived class inherits base class members there is a chance of base class data member are similar to derived class data member and JVM gets an ambiguity or confusion.

Syntax:

```
Super.baseclass datamember name
```

For example:

```
class VarTest
{
float salary=1000;
}
class Derived extends VarTest
{
float salary=2000;
void display()
{
System.out.println("Salary: "+super.salary);//print base class salary
}
}
class Supervarible
{
public static void main(String[] args)
{
```

```
Derived obj=new Derived();
obj.display();
}
}
```

Super at method level

The **super keyword** can also be used to invoke or call base class method. It should be used in case of method overriding. In other words **super keyword** use when base class method name and derived class method name have same name.

For example:

```
class Student
{
void message()
{
System.out.println("Good Morning to all");
}
}
class Faculty extends Student
{
void message()
{
System.out.println("Good Morning Students ….");
}
void display()
{
```

```
message();//will invoke or call current class message() method
super.message();//will invoke or call parent class message()
method
}
public static void main(String args[])
{
Student s=new Student();
s.display();
}
}
```

Super at constructor level

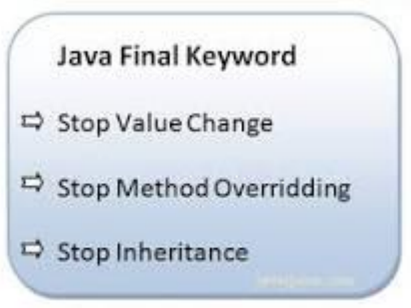
The super keyword can also be used to invoke the base class constructor. Constructors are calling from bottom to top and executing from top to bottom manner.

Example:

```
class Employees
{
Employees()
{
System.out.println("Employees class Constructor");
}
}
class HRs extends Employee
```

```
{
HRs()
{
super(); //will invoke or call parent class constructor
System.out.println("HR class Constructor");
}
}
class Supercons
{
public static void main(String[] args)
{
HRs obj=new HRs();
}
}
```

FINAL KEYWORD:



The final keyword is used with the variables, a final variable that has no value it is called blank final variable or an uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which can be initialized in the static block only.

The final keyword in java is used to restrict the user. The java final keyword can be used in many contexts. Final can be:

1. variable level
2. method level
3. class level

Let's first learn the basics of final keyword given below.

1) Java final variable

If you declare any variable as final, you cannot change the value of final variable (It will be like constant). There is a final variable speed limit, we are going to change the value of this variable, but it can't be changed because final variable once assigned a value can never be changed in any way.

```
class Bikes{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bikes obj=new  Bikes();
 obj.run();
 }
}//end of class
```

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
class Bikess{
  final void run(){
System.out.println("running fastly");
}
}

class Honda extends Bikess{
   void run(){System.out.println("running safely with 105kmph …");}
   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
final class Bikes{
}
```

```
class Honda1 extends Bikes{
  void run()
{
  System.out.println("running safely with 100kmph");
}
  public static void main(String args[]){
  Honda1 honda= new Honda();
  honda.run();
  }
}
```

STATIC KEYWORD:

The static keyword is used for memory management mainly in java. Memory Management plays a vital role in programming language. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class and can be given below in detail.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

1) Java static variable

If you declare any variable as static, it is known static variable.

- The static variables can be used to refer the common property of all objects e.g. company name of employees, college name of students etc.
- The static variable gets memory only once in class area at the time of class loading.

Advantage of static variable:

It makes your program memory efficient (i.e it saves memory).

Understanding problem without static variable

```
 class Student{
 int rollno;
 String name;
 String college="JNTUA";
}
```

Suppose there are 500 students in my college, now all instance data members will get memory each time when object is created. All students have unique roll no and name so instance data member is good. Here, college refers to the common property of all objects. If we make it static, this field will get memory only once.

Example of static variable

```
 class Student{
int rollno;
String name;
static String college ="JNTUA";
Student(int r,String n){
rollno = r;
name = n;
}
void display (){System.out.println(rollno+" "+name+" "+college);}

public static void main(String args[]){
Student s1 = new Student(5301,"JP");
Student s2 = new Student(5306,"ASR");
s1.display();
s2.display();
}
}
```

3.6 PARAMETER PASSING:

Java does access objects by reference, and all object variables are references in it. However, Java doesn't pass method arguments by reference that is statically; it passes them by value. The method successfully alters the value of pntx1, even though it is passed by value; however, a swap of pntx1 and pntx2 fails!

3.7 RECURSION:

Recursion is a basic programming technique and can be used in Java, in which a method calls itself to solve some problem. A method that uses this technique is recursive. Many programming problems can be solved only by recursion, and some problems that can be solved by other techniques are better solved by recursion. So that the same code can be used repeatedly.

A function or a method which calls itself is known as recursion and the function is called as recursive function.

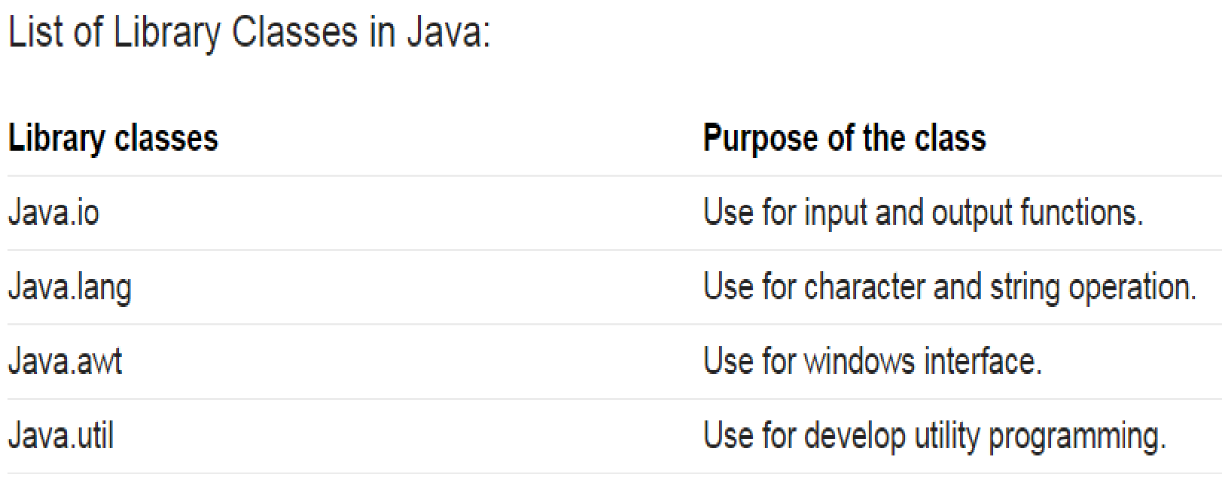
Program:

```
class Factorial
{
int fact(int x)
{
if (x==1)
return 1;
else
return (fact(x-1) * x);
}}
class Recursion
{
public static void main(String args[])
{
Factorial f = new Factorial();
System.out.print("Factorial of 6 is ");
System.out.println(f.fact(6));
} }
```

OUTPUT:

```
Factorial of 5 is: 720
```

3.8 THE IMPORTANT JAVA CLASS LIBRARIES:



List of Library Classes in Java:

Library classes	Purpose of the class
Java.io	Use for input and output functions.
Java.lang	Use for character and string operation.
Java.awt	Use for windows interface.
Java.util	Use for develop utility programming.

3.9 ARRAYS:

An Array is a group of similar Data type items that share a common name. Array size is fixed and cannot be altered dynamically. In Arrays, Memory is allocated using the new operator, Array can be declared as follows

datatype variable[] = new datatype[size];

Creation of array involves three steps

- Declaring the array.
- Creating the array memory location.
- Assigning values to the memory location.

Arrays creation can be of two types

- Single-Dimensional Array.
- Multi-Dimensional Array.

Declaring the **Single Dimensional Array** can be done in two way.

Ex :

int number[ ];

Int[ ] number;

We should not mention the size of the array at Declaration.

Creating Single-Dimensional Array in Memory Location can be done as.

Ex :

int number[ ];

//Declaration of array

Number = new int[5];

//Creation of array

Elements are 5, but their index numbers are 0 to 4.

Assigning Single-Dimensional Array values in the memory locations can be done as.

Ex:

number[0] = 3;

number[1] = 1;

number[2] = 17;

number[3] = 5; …..

// Single-Dimensional Array Example.

Program:

```
Class SingleDimensionArray
{
public static void main(String args[ ])
{
// Declare and allocate space
int myarray[ ] =new int[4];
// Initialize elements
myarray[0] = 3;
myarray[1] = 1;
myarray[2] = 2;
myarray[3] = 5;
// Display length
System.out.println("myarray.length = " +myarray.length);
// Display elements
System.out.println(myarray[0]);
System.out.println(myarray[1]);
System.out.println(myarray[2]);
System.out.println(myarray[3]);
}}
```

Multidimensional arrays are arrays of arrays. Declaring the Single Dimensional Array can be done in two
ways.

Ex :

```
int number[ ][ ];
Int[ ][ ] number;
```

Creating Multi-Dimensional Array in Memory Location can be done as…

```
int number[ ][ ];
//Declaration
Number = new int[ 4 ][ ];
//Creation
```

Here the second array need not be specified.

Assigning Multi-Dimensional Array values in the memory locations can be done as.

Ex:

```
number[0][0] = 33;
number[0][1] = 71;
number[1][0] = 16;
number[1][1] = 45;

class TwoDimension
{
public static voidmain(String args[ ])
{
// Declare and allocate space
int myarray[ ][ ] = new int[3][2];
// Initialize elements
myarray[0][0]= 33;
myarray[0][1] = 71;
myarray[1][0] = 16;
myarray[1][1] = 45;
myarray[2][0]= 9;
myarray[2][1] = 2;
// Display length
System.out.println("myarray.length = " + myarray.length);
// Display elements
System.out.println(myarray[0][0]);
System.out.println(myarray[0][1]);
System.out.println(myarray[1][0]);
System.out.println(myarray[1][1]);
System.out.println(myarray[2][0]);
System.out.println(myarray[2][1]);
}}
```

3.9.1 STRINGS: A string is a sequence of characters enclosed in double quotes.

String Methods

Here is the list of methods supported by String class in java−

1 char charAt(int index)

Returns the character for specified index.

2 int compareTo(Object o)

Compares this String to another Object.

3 int compareTo(String anotherString)

Compares two strings lexicographically.

4 int compareToIgnoreCase(String str)
Compares two strings lexicographically, ignoring case differences.

5 String concat(String str)
Concatenates the specified string to the end of this string.

6 boolean contentEquals(StringBuffer sb)
Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.

7 static String copyValueOf(char[] data)
Returns a String that represents the character sequence in the array specified.

8 static String copyValueOf(char[] data, int offset, int count)
Returns a String that represents the character sequence in the array specified.

9 boolean endsWith(String suffix)
Tests if this string ends with the specified suffix.

10 boolean equals(Object anObject)
Compares this string to the specified object.

11 boolean equalsIgnoreCase(String anotherString)
Compares this String to another String, ignoring case considerations.

12 byte getBytes()
Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.

13 byte[] getBytes(String charsetName)
Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.

14 void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
Copies characters from this string into the destination character array.

15 int hashCode()
Returns a hash code for this string.

16 int indexOf(int ch)
Returns the index within this string of the first occurrence of the specified character.

17 int indexOf(int ch, int fromIndex)

Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

18 int indexOf(String str)

Returns the index within this string of the first occurrence of the specified substring.

19 int indexOf(String str, int fromIndex)

Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

20 String intern()

Returns a canonical representation for the string object.

21 int lastIndexOf(int ch)

Returns the index within this string of the last occurrence of the particular character.

22 int lastIndexOf(int ch, int fromIndex)

Returns the index within the string of the last occurrence of the specified character, searching backward starting at the specified index.

23 int lastIndexOf(String str)

Returns the index within this string of the rightmost occurrence of the particular substring.

24 int lastIndexOf(String str, int fromIndex)
Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.

25 int length()
Returns the length of this string.

26 boolean matches(String regex)
Tells whether or not this string matches the given regular expression.

27 boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)
Tests if two string regions are equal.

28 boolean regionMatches(int toffset, String other, int ooffset, int len)
Tests if two string regions are equal.

29 String replace(char oldChar, char newChar)

Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

30 String replaceAll(String regex, String replacement

Replaces each substring of this string that matches the given regular expression with the given replacement.

31 String replaceFirst(String regex, String replacement)

Replaces the first substring of this string that matches the given regular expression with the given replacement.

32 String[] split(String regex)

Splits this string around matches of the given regular expression.

33 String[] split(String regex, int limit)

Splits this string around matches of the given regular expression.

34 boolean startsWith(String prefix)

Tests if this string starts with the specified prefix.

35 boolean startsWith(String prefix, int toffset)

Tests if this string starts with the specified prefix beginning a specified index.

36 CharSequence subSequence(int beginIndex, int endIndex)

Returns a new character sequence that is a subsequence of this sequence.

37 String substring(int beginIndex)
Returns a new string that is a substring of this string.

38 String substring(int beginIndex, int endIndex)
Returns a new string that is a substring of this string.

39 char[] toCharArray()
Converts this string to a new character array.

40 String toLowerCase()
Converts all of the characters in this String to lower case using the rules of the default locale.

41 String toLowerCase(Locale locale)
Converts all of the characters in this String to lower case using the rules of the given Locale.

42 String toString()
This object (which is already a string!) is itself returned.

43 String toUpperCase()

Converts all of the characters in this String to upper case using the rules of the default locale.

44 String toUpperCase(Locale locale)

Converts all of the characters in this String to upper case using the rules of the given Locale.

45 String trim()

Returns a copy of the string, with leading and trailing whitespace omitted.

46 static String valueOf(primitive data type x)

Returns the string representation of the passed data type argument.

3.9.2 TYPE CONVERSION AND CASTING:

If the two types are compatible, then Java will perform the conversion automatically. For example, assign an int value to a long variable. For incompatible types we must use a cast. Casting is an explicit conversion between incompatible types.

Java's Automatic Conversions: An automatic type conversion will be used if the following two conditions are met:

The two types are compatible.

The destination type is larger than the source type of value.

int type is always large enough to hold all valid byte values, so an automatic type conversion takes place.

```
public class Main {
  public static void main(String[] argv) {
    byte b = 20;
    int i = 0;

    i = b;
    System.out.println("value of b is " + b);
    System.out.println("value of i is " + i);
  }
}
```

For **widening conversions**, integer and floating-point types are compatible with each other.

```
public class Main {
  public static void main(String[] argv) {
    int i = 456;
    float f;
    f = i;
    System.out.println("value of i is " + i);
    System.out.println("value of f is " + f);
  }
}
```

3.9.3 GARBAGE COLLECTION:

In general memory is allocated to an object by using “NEW “ keyword and deallocation of memory was done by using “DELETE‟ keyword in C++. But in java the allocation of memory for an object is done by “NEW‟ operator but deallocation is done automatically which is done by garbage collector algorithm. And the method which is used by garbage collector is gc().

CHAPTER 4
INHERITANCE

4.0 Introduction:

Inheritance is the process of acquiring the properties of a base class object to the properties of the derived class object.

Java doesn't support multiple inheritance but to use this concept interfaces must be used.

This concept provides the idea of CODE REUSABILITY in OOPS. For acquiring properties from one class to another class **extends** keyword is used.

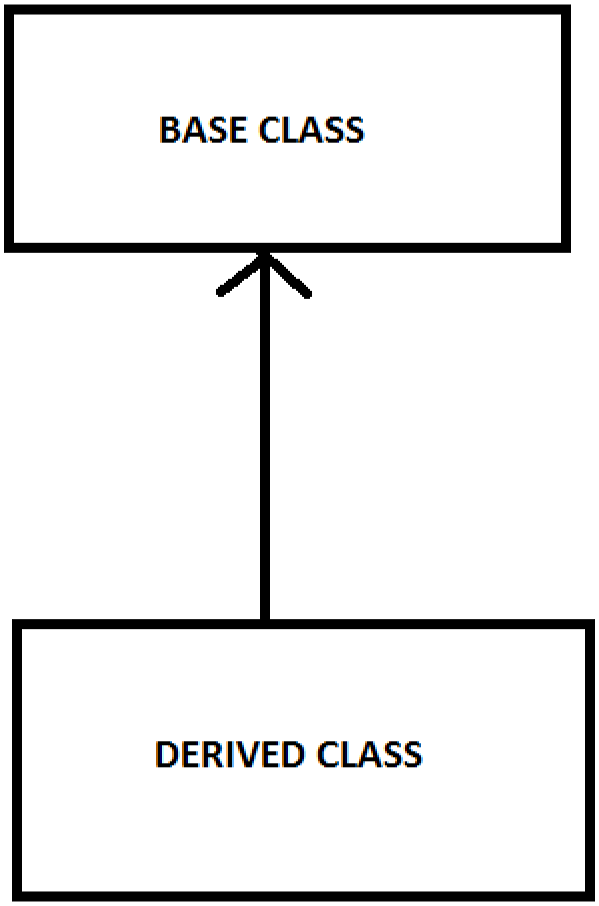
4.1 Types of Inheritance:

There are 4 types of inheritances. They are

- Single inheritance
- Multilevel inheritance
- Multiple inheritance
- Hierarchal inheritance

Single Inheritance:

It is the process of acquiring properties from base class to the derived class.



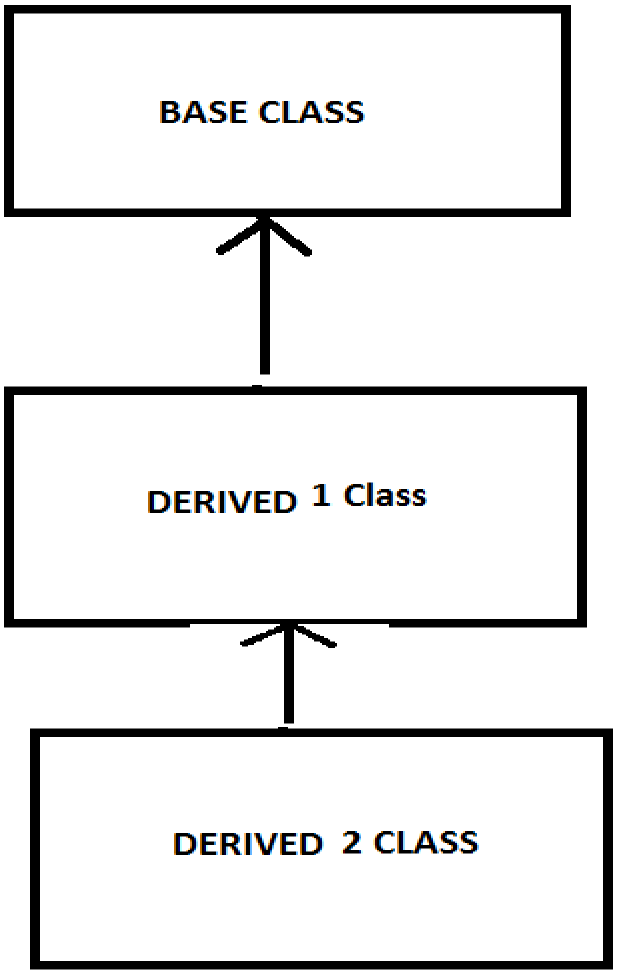
For example,

```
class Base
{
protected:
int a,b;
Base()
{
```

```
a=100;
b=20;
}
};
class Derived extends Base
{
public:
void add();
void mul();
};
Void Derived::add()
{
System.out.println("sum of two variables is:"+(a+b));
}
Void Derived::mul()
{
System.out.println("Product of two variables is:"+(a*b));
}
void main()
{
public static void main(String args[])
{
clrscr();
Derived obj=new Derived();
obj.add();
obj.mul();
getch();
}
```

Multilevel Inheritance:

It is the process which acquires the properties from already existing derived class to another derived class.



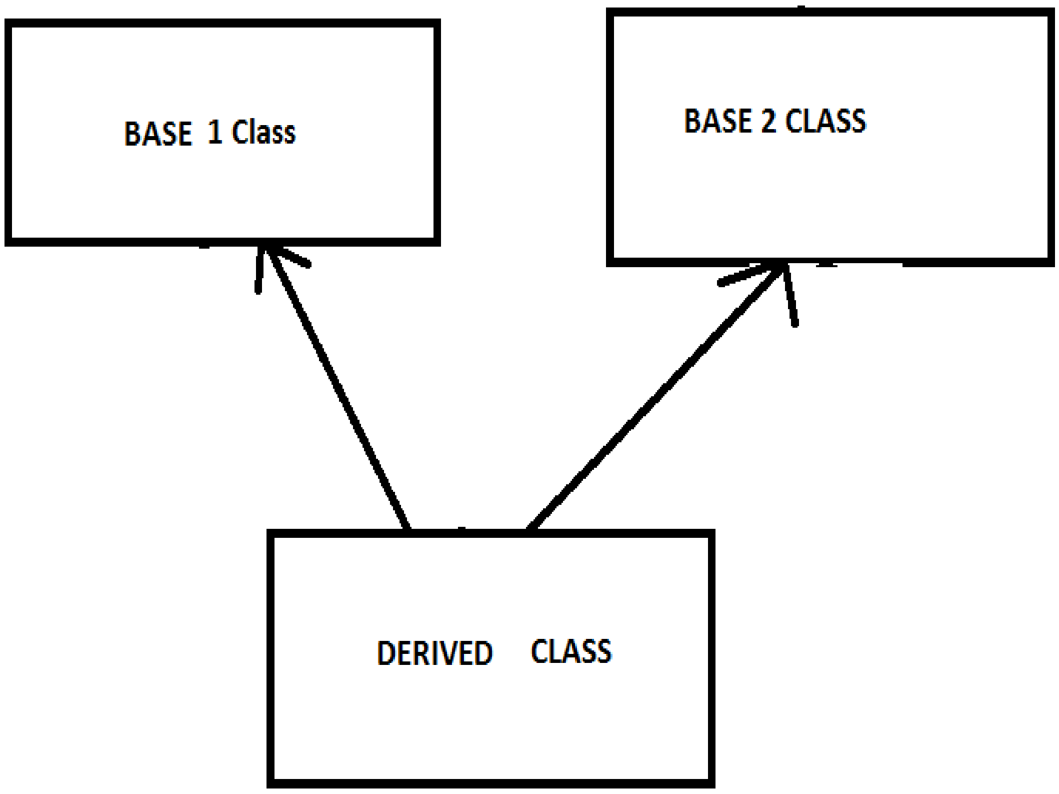
For example,

```
class Base
{
protected:
int a;
BASE()
{
a=100;
}
};
class Derived1 extends Base
{
protected:
int b;
Derived1()
{
b=20;
}
};
class Derived2 extends Derived1
{
public:
void add();
void mul();
};
```

```
Void Derived2::add()
{
System.out.println(“sum of two variables is:”+(a+b));
}
Void Derived2::mul()
{
System.out.println(“Product of two variables is:”+(a*b));
}
void main()
{
public static void main(String args[])
{
clrscr();
Derived2 obj=new Derived2();
obj.add();
obj.mul();
getch();
}
```

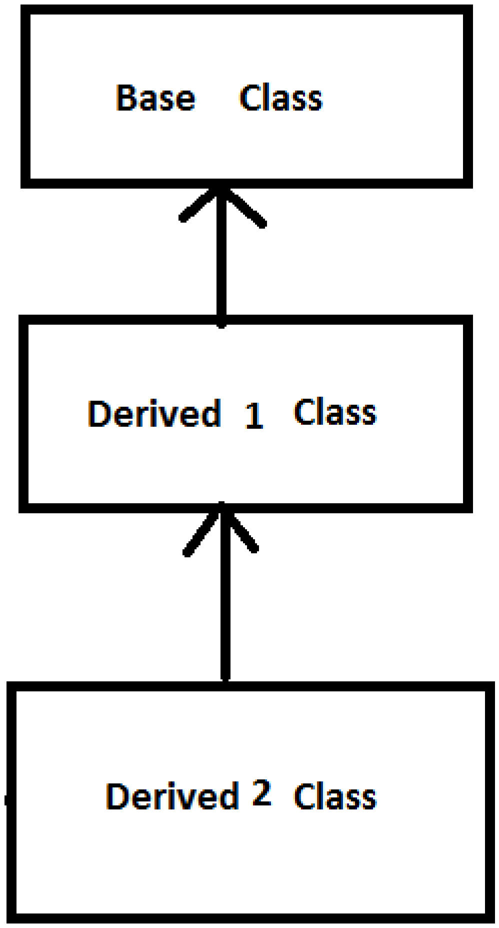
Multiple Inheritance:

This inheritance occurs when one derived class acquires the properties from two base classes. This inheritance can be implemented using interface concept and will be explained in interfaces chapter.



Hierarchal Inheritance:

It occurs when one derived class acquires the properties from two base classes.



For example,

```
class Base
{
protected:
int a;
Base()
{
a=100;
}
```

```
};
Class Derived1 extends Base
{
Protected:
Public:
Int b;
Public:
Derived1()
{
b=20;
}
};
class Derived2 extends  Derived1
{
public:
void add();
void mul();
};
Void Derived2::add()
{
System.out.println("sum of two variables is:"+(a+b));
}
Void Derived2::mul()
{
System.out.println("Product of two variables is:"+(a*b));
```

```
}
void main()
{
public static void main(String args[])
{
clrscr();
Derived2 obj=new Derived2();
obj.add();
obj.mul();
getch();
}
```

4.2 Forms of Inheritance:

- Specialization
- Specification
- Construction
- Extension
- Limitation
- Combination

Specialization:

The child class is a special class of parent class that is the child class is a subtype of parent class.

Specification:

The parent class defines behavior that is implemented in the child class but not in the parent class.

Construction:

The child class makes use of behavior provided by parent class but it is not a subtype of parent class.

Extension:

The child class adds new functionalities to the parent class but doesn’t change any inherited behavior.

Limitation:

The child class restricts the use of sum of the behaviors inherited from the parent class.

Combination:

The child class inherits features from more than one parent class, although multiple inheritance is not supported by java.

4.3 Benefits of inheritance:

- Reusability
- Reliability
- Code sharing
- Consistency of interface
- Rapid Prototyping
- Information Hiding
- Software components
- Polymorphism.

4.4 Costs of inheritance:

- Speed of Program execution
- Size of the program
- Message passing overhead
- Program Complexity

Member access rules:

There are different member access rules like public, private, and protected. Public is used to make a class visible to all members in a program, whereas private is used for making members, private to the particular class. And protected used in the concept of inheritance.

4.5 Using super and final with inheritance:

Example:

```
Class Employees
{
Employees()
{
System.out.println("Employees class Constructor");
}
}
class HRs extends Employee
{
HRs()
{
```

```
super(); //will invoke or call parent class constructor
System.out.println("HR class Constructor");
}
}
class Supercons
{
public static void main(String[] args)
{
HRs obj=new HRs();
}
}
```

Using Final: Showing usage of final keyword

If we declare a method as final, then that method cannot be overridden. And also if we declare a variable as final then that value cannot be changed.

```
class A
{
final void meth()
{
System.out.println("This is a final method.");
}}
class B extends A
 {
void meth() // ERROR! Can't override. It can be illegal
{
```

```
System.out.println("Illegal!");
}
}
```

4.6 Polymorphism:

This word is derived from Greek word. Poly means many and morphos means forms. This means having more than one form. Method overloading means writing same name for method and using different number of parameters.

For example Method Overloading as follows:

```
class DisplayOverloading
{
   public void disp(char c)
   {
       System.out.println(c);
   }
   public void disp(char c, int num)
   {
       System.out.println(c + " "+num);
   }
}
class Sample
{
  public static void main(String args[])
  {
     DisplayOverloading obj = new DisplayOverloading();
```

```
        obj.disp('a');
        obj.disp('a',120);
   }
}
```

4.7 Method Overriding: It means writing a method in a base class that is rewritten in the derived class.

```
class A
{
void meth()
{
System.out.println("This is a final method.");
}}
class B extends A
 {
void meth()
{
System.out.println("legal! Can be overriden");
}
}
```

4.8 Abstract Classes:

Abstract class is the class that which has at least one pure virtual function. A pure virtual function is said to have a function declared within a base class with a value zero.

For Example:

```
Abstract Class Base{
```

```
Protected:
Int a,b;
Public:
Base()
{
A=100;
B=20;
}
Virtual int add()=0; or int add();
}
Class Derived extends Base
{
Public:
Int add()
{
Return (a-b);
}
}
Class AbstClassTest
{
Public static void main*(String args[])
{
Derived obj=new Derived();
System.out.println(“ The value is:”+obj.add());
}}
```

CHAPTER 5

PACKAGES AND INTERFACES

5.0 Packages – Introduction:

A Package is a collection of classes, methods and interfaces which can be defined, created and accessed in other new programs in order to increase code extensibility & reusability features in object oriented programming.

Packages can be predefined or user defined. This will helps programmers to write programs easily by using the existing code by importing packages.

A Package is defined as collection of classes, interfaces and sub packages. A class is defined as collection of variables and methods. The variables can be either static variables or instance variables and the methods can be either instance methods or static methods. As we know that, encapsulation is a concept in Object Orientation Paradigm, which is technically implemented with the help of classes and interfaces. An interface is defined as a construct that consists of variables and methods. The variables with in a interface is known as interface variables and the methods in an interface is known as abstract methods. A sub- package is defined as a package present with in a package. It also contains classes and interfaces.

5.1 Types of Packages:

They are two types of packages that exist in Java programming language. They are:

1. Predefined Package
2. User Defined Package

Predefined Package:

It is defined as a package whose classes and interfaces are already defined by the Java Developers. These packages are defined already by the developers keeping in mind of the programmers that can use the methods present in the classes and interfaces of a particular package whenever it is required. For example if the programmer is designing an application that require connectivity to the database, then we require some methods that can be used to access or retrieve the data from the database and some methods that are used to store the data into database. These methods are present in the class that is present in a predefined package known as java.sql. In other words, if user is writing a program on JDBC, then the user need not bother about what methods those are required but just use the package java.sql by importing it within the program. Likewise, Java developers have developed some packages that can be used whenever it is required by their application or program.

There are many predefined packages, but the most important are:

1. java.lang
2. java.awt
3. java.net
4. java.sql
5. java.util
6. javax.swing

User defined Packages:

A user defined package is defined as a package, whose classes and interfaces have to be written by the users or the programmers. In other words, as we know a class consists of concrete methods where as interface consists of abstract methods. In case of User Defined Package, in case of classes, it is the responsibility of the user to write the functionality of concrete methods, and in case of interface it is responsibility of the user is to write the name of the abstract method.

5.2 Creating a Package:

To create a user-defined package, a keyword package is used as shown below:

```
package packagename;
```

It must be noted that whenever user wants to write a class with in a package, then with in a notepad or editor we can write only one class at a time. Writing more than one class with in a notepad will lead to compile time error, when the first statement is package creation statement. But, if we are omitting that statement, then no compile time error gets generated. The syntax to write a class with in a package is as shown below:

```
package packagename;
class classname{
---------------------
---------------------
---------------------
}
```

The above code can be written either in a notepad or java editor or any editor. It must be noted that the name of the notepad or file, must be same as the name of the class that is present within the file. This class must be a single class i.e. if we write more than one class with in a same file, then a compile time error gets generated. But, if user wants to include one more class with in a package, then the user has to open new notepad or java editor, write the first statement as package creation statement , followed by the name of the class and save the name of the file as same as the name of the class.

A Java program that is used to create a package that is having only two classes.

Step1: Creating Addition class in a package known as student.

```
package student;
public class Addition
{
int i,j;
public Addition(int a,int b)
{
i=a;
j=b;
}
public void display()
{
System.out.println("The added value is:"+(i+j));
} }
```

Step2: Creating Substraction class in the same package known as student.

```
package student;
public class Substraction
{
int i,j;
public Substraction(int a,int b)
{
i=a;
j=b;
}
public void display()
{
System.out.println("The added value is:"+(i-j));
} }
```

The above step1 must be written in separate notepad and the step2 must also written in separate notepad. In the above step1 and step2 we can notice that when user is writing a class with in a package, then it is compulsory that the class must be public, the methods with in the class must be public, the constructors with in the class must be public.

5.3 Compilation of Packages & Understanding CLASSPATH

The compilation of a package is different from the compilation of a java
program. The syntax to compile a package is shown below:

```
javac –d <directory path> Classname.java
```

-d: it is an option of javac, that is used to create a directory or a folder.

<directory path> : It includes the location at which the directory must be stored. In other words, if there is (.) then it represents current working directory or present working directory.

When user press enter key, then a directory with the name that is present in the package gets created and this directory is stored in a location that has been specified in the directory path and the. class file is stored in this directory.

5.3.1 Accessing a Package:

If user wants to access a package, then the user has to write a java program that can access the methods that are present in the class that are present in the package. This java program, must contain a statement that is used to access a package. The statement is nothing but import statement. With the help of import statement, the java program can access the methods that are present with in the class and these classes are present in the package.

A java program that is used to access a package having only two classes.

```
import student.Addition;
import student.Substraction;
class Pack{
public static void main(String sun[]){
Addition a1 = new Addition(2,3);
a1.display();
Substraction s1 = new Substraction(10,4);
s1.display();
}
}
```

The above java program is written in a notepad, whose name is same as the name of the class. Here the class name is Pack i.e. the name of the notepad is Pack.java . When the above program gets compiled with the help of javac command, a .class file gets generated and when the above program gets executed with the help of java command, then we can see the output.

5.3.2 Importing & Understanding Package:

If the programmer wants to access a package, whether the package is user defined or predefined, then it is possible with the help of a statement known as import statement. They are two types of import statements in Java:

1. Implicit import or static import
2. Explicit import or general import

Implicit import or static import:

The syntax of implicit import statement is

import packagename.*;

If the programmer wants to include all classes and interfaces of a subpackage then the syntax is

import packagename.subpackagename.*;

If the programmer writes implicit import statement, then during the execution of the program, it is the responsibility of the JVM to load all the classes and interfaces that are present with in a package only. It is to be made it clear that, classes and interfaces of sub packages are not loaded into the main memory. Though the user or programmer requires only two classes and no interfaces in their program also, the JVM will load all classes and interfaces. This lead to program to increase the efficiency of the program and the fast execution of the program, it is recommended that the programmer use explicit import. More over this type of import statements are only used in academics but never used in real time.

Let us consider an example to understand the concept of Implicit import. The following examples are:

```
import java.awt.*;
```

When the above statement gets executed, all classes and interfaces of package awt are loaded into the main memory, though the user is interested only in some methods or some classes.

```
import java.awt.event.*;
```

When the above statement gets executed, all classes and interfaces present in the sub-package event are loaded in the main memory, though the user is interested only in some methods or some classes. As in the above cases, we can notice that the classes and interfaces that the user is not interested are also getting loaded into main memory. (It must be noted that the word interested reflects to the methods that are present in the classes And interfaces that are not required to the program or application that user is writing).

So, if the programmer is interested only in loading of the methods (Present in the classes and interfaces), that are present in the application, and if the programmer wants no delay during the execution of the program, then it is advisable to go for explicit import or general import.

Explicit import or general import:

The syntax of explicit import statement is

import packagename. classname.methodname;

If the programmer wants to include all classes and interfaces of a sub package then the syntax is

import packagename.subpackagename. classname.methodname;

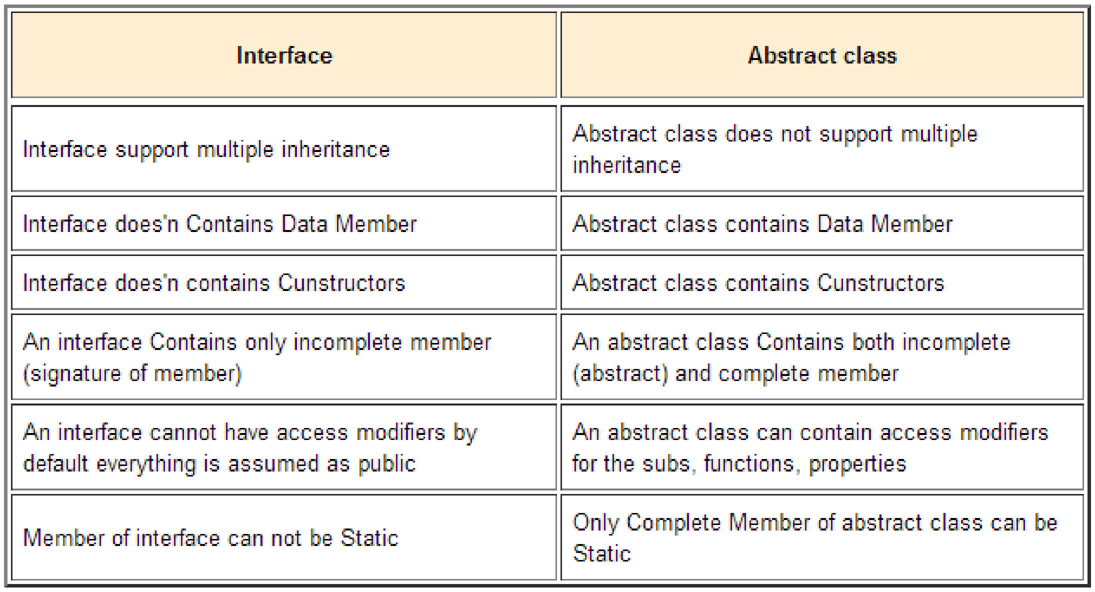
If the programmer writes explicit import statement, then during the execution of the program, it is the responsibility of the JVM to load only the method that is specified in the statement. In other words, in case of explicit import other methods that are not specified are not loaded into main memory. In this case there is no delay in execution of the program and the efficiency of the program also increases. More over this type of import statements are only used in real time.

Let us consider an example to understand the concept of implicit import. The following examples are:

import java.awt.event.ActionEvent;

When the above statement gets executed, only the methods that are present in the class ActionEvent are loaded on to the main memory. So in this way we can save the time i.e. the execution time and this increases the efficiency of the application or the program that is developed by the user. .all classes and interfaces present in the sub-package event are loaded in the main memory.

5.4 Differences between abstract classes and interfaces:



Interface	Abstract class
Interface support multiple inheritance	Abstract class does not support multiple inheritance
Interface does'n Contains Data Member	Abstract class contains Data Member
Interface does'n contains Cunstructors	Abstract class contains Cunstructors
An interface Contains only incomplete member (signature of member)	An abstract class Contains both incomplete (abstract) and complete member
An interface cannot have access modifiers by default everything is assumed as public	An abstract class can contain access modifiers for the subs, functions, properties
Member of interface can not be Static	Only Complete Member of abstract class can be Static

5.5 Defining an interface:

As we have studied a concept known as Encapsulation in Object Oriented Paradigm. It states that wrapping of variables and methods into one unit. This is done just to provide security to data that is present in variables and security to information that is present in the method. Technically, the concept of encapsulation is implemented with the help of classes and interfaces. As we know that a class is defined as a collection of variables and methods. The methods that are present in a class is known as concrete methods.

An interface is defined as collection of abstract methods and interface variables. If an interface contains concrete methods, then java compiler generates a compile time error. An abstract method is defined as a method that does not contain body. In other words an interface consists of only method prototypes. Since, these methods are incomplete methods, we cannot create objects for interfaces. The methods present in the interface are public and abstract. In other words, the access specifier that is used for the methods present in an interface is public, so that these interfaces can be accessed from anywhere. The keyword abstract, says java compiler that these methods don't have body. These abstract methods present in the interfaces are terminated by semicolon. If semicolon is missing, then we get a compile time error.

The syntax to create an interface is:

```
interface interface_name
{
public abstract return_datatype methodname();
------------------------------
-
------------------------------
}
```

There can exist only one abstract method or any number of abstract methods. Number of abstract methods in an interface depends on the type of program, a programmer is writing.

Implementation Class The class that take the responsibility of defining the body of the abstract methods in an interface is known as implementation class. Moreover, this class can define their methods of their own. The syntax of implementation class is

```
class ClassName implements interfaceName{
------------------------------
----------
------------------------------
----------
------------------------------
----------
------------------------------
----------
}
```

An implementation class can be defined as a class that contains keyword as implement followed by the name of an interface. It is mandatory that any class, which acts as implementation class must define bodies of all abstract methods that are present in the interface, for which the class has taken responsibility. Let us understand with the help of an example. If an interface has 3 abstract methods, and if the implementation class has define only

2 abstract methods, then during compilation of the java program, Java Compiler will generate compile time error.

5.5.1 Interface Reference:

As we have studied that it is not possible to create objects to an interface. So, we can create a reference to an interface. This reference is known as Interface Reference. An interface reference is written in an implementation class. An interface reference has ability to access only those methods that are present within the interface but cannot access those methods that are not present in the interface but defined in implementation class. In this case, the programmer has to create an object that has ability to access any method present within the implementation class. Any method means whether the method is present in the interface or not.

The syntax of interface reference is:

```
interfacename interfacereference = new
ConstructorofImplementationclass
```

A program to create an interface and access this interface using interface reference.

```
interface Inter
{
void addTwo(int a,int b);
void addThree(int a,int b,int c);
}
```

```
class Arith implements Inter
{

public void addTwo(int a,int b)
{
System.out.println("The result of two values "+(a+b));
}
public void addThree(int a,int b,int c)
{
System.out.println("The result of three values "+(a+b+c));
}
}
class Arithmetic
{
public static void main(String ag[ ])
{
Inter i = new Arith();
i.addTwo(2, 3);
i.addThree(6, 5, 9);
}
}
```

5.5.2 Interface Variables:

Variables present in a class are known as instance variables, where as variables present in an interface is known as interface variables. The syntax of interface variables is :

```
public final static datatype variablename;
```

It must be noted that interface variables can be accessed without creating either object or interface reference because interface variables are static by nature. If programmer wants to change the value of interface variable with an implementation class, then the java compiler will generate a compile time error because interface variables are final in nature. To be more specific, interface variables are by default public, final and static. In other words, if the programmer as not mention public, static and final keywords while declaring a variable, then the java compiler will not give any compile time error. a program to illustrate the concept of interface variables

```
interface InterfaceVariables
{
int x=10;
}
class Interf1 implements InterfaceVariables
{
public static void main(String args[]){
Interf1 in = new Interf1();
System.out.println(in.x);
//in.x=112;
} }
```

When the above program gets executed, we get output as 10. But, if we remove the single comment, then if we save and then compile the program, we get compile time error as saying that we cannot assign a value to variable x.

5.5.3 Extended Interface:

We can inherit an interface from another interface. The syntax holds good as same as the class. When a class implements an interface, that inherits another interface, it becomes mandatory for the implementation class to implement the methods that are present in the interface and the methods that have been inherited from other interface. With this concept, we can implement Multiple inheritance in indirect way.

A program that illustrates the concept of extension of the interface.

```
interface A
{
void d1();
}
interface B extends A
{
void d2();
}
class C implements B
{
public void d1( )
{
System.out.println("Method of Interface A is called");
}
public void d2( )
{
System.out.println("Method of Interface B is called");
} }
class InterfEx
{
public static void main(String args[ ])
{
C c1 = new C( );
c1.d1( );
c1.d2( );
}
}
```

In the above program, we can notice that, there are two interfaces namely A and B. We can notice that interface A contains only one method. There is another interface by name B, which has two methods one of its own and the another method is inherited from the interface A as we are using a keyword known as extends. After writing the interface, we are creating a class known as implementation class, whose responsibility is to write the body of the abstract methods that are declared within the interface. It is mandatory that this implementation class must contain the bodies of the abstract methods that are present in the interface B. If any one of the method body is missing, then a compile time error gets generated. We are creating one more class that contains main method. In this method, we are creating objects of implementation class so that we can call the methods that are present in the implementation class.

CHAPTER 6

EXCEPTION HANDLING

6.0 Introduction to Exception Handling Fundamentals:

An exception is an event that arises during the execution of a program. When an Exception occur 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.

6.1 Exception Types:

An exception can occur for many different reasons. Following are some of the scenarios where an exception can be occurred.

- If a user has entered an invalid data.
- If a file that needs to be opened cannot be found.
- And network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner/way.

Based on these, we have three kinds of Exceptions. We have to understand them to know how exception handling works in Java.

Checked exceptions − A checked exception is an exception that occurs at the compile time, 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.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

Example

```
import java.io.File;
import java.io.FileReader;
public class FilenotFound_Demo {
   public static void main(String args[]) {
      File file = new File("E://files.txt");
      FileReader fr = new FileReader(files);
   }
}
```

If you try to compile the above program, you will get the following exceptions.

Output

```
C:\>javac FilenotFound_Demo.java
FilenotFound_Demo.java:8: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
      FileReader fr = new FileReader(files);
                      ^
1 error
```

Note − Since the methods **read()** and **close()** of FileReader class throws IOException, you can observe that the compiler notifies to handle IOException, along with FileNotFoundException.

Unchecked exceptions − an unchecked exception is an exception that occurs at the time of execution that is runtime. These are also called as **Runtime Exceptions**. These include programming bugs or we call as errors, such as logic errors or improper use of an API. Runtime exceptions are ignored/skipped at the time of compilation process.

For example, if you have declared an array of size 5 in your program and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsExceptionexception* occurs.

Example

```
public class Unchecked_Demo {
  public static void main(String args[]) {
    int num[] = {1, 2, 3, 8};
    System.out.println(num[5]);
  }
}
```

If you compile and execute the above program, you will get the following exception.

Output

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 5
at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)
```

Errors − are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will get. They are also ignored at the time of compilation process.

Uncaught Exceptions:

The java.lang.ThreadGroup.uncaughtException() method is called by the Java Virtual Machine when a thread in this thread group stops because of an uncaught exception, and the thread does not have a specific Thread.

Declaration

Following is the declaration for java.lang.ThreadGroup.uncaughtException() method

```
public void uncaughtException(Thread t, Throwable e)
```

Parameters

- t -- This is the thread that is about to exit.
- e -- This is the uncaught exception.

Return Value

This method does not return any value.

Exception

- NA

Example

The following example shows the usage of java.lang.ThreadGroup.uncaughtException() method.

```
package com.jp;

import java.lang.*;

 public class ThreadGroupDemo implements Runnable {

    public static void main(String[] args) {
      ThreadGroupDemo tg = new ThreadGroupDemo();
      tg.func();
   }
   public void func()
   {
      try {
         // create a new ThreadGroup and a child for that
ThreadGroup.
         newThreadGroup pGroup = new
newThreadGroup("ParentThreadGroup");
         newThreadGroup cGroup = new newThreadGroup
         (pGroup,"ChildThreadGroup")
         // create another thread
         Thread thr2 = new Thread(pGroup, this);
         System.out.println("Starting " + thr2.getName() + "...");
         // this will call run() method
         thr2.start();
```

```
// create third thread

Thread thr3 = new Thread(cGroup, this);

System.out.println("Starting " + thr3.getName() + "...");

// this will call run() method

thr3.start();


try {

  Thread.sleep(500);

}

catch(InterruptedException ex) {}

// interrupt the two threads

thr2.interrupt();

thr3.interrupt();


// block until the other threads finish

thr2.join();
```

```
      thr3.join();
    }
    catch(InterruptedException e) {
      System.out.println(e.toString());
    }
}


public void run() {
    try {
      System.out.print(Thread.currentThread().getName());
      System.out.println(" executing...");


      while(true) {
        Thread.sleep(500);
      }
    }
```

```
        catch(InterruptedException e) {
           Thread currThread = Thread.currentThread();
           System.out.print(currThread.getName());
           System.out.println(" interrupted:" + e.toString());
           // rethrow the exception
           throw new RuntimeException(e.getMessage());
        }
    }
}
class newThreadGroup extends ThreadGroup {
  newThreadGroup(String n) {
      super(n);
   }
   newThreadGroup(ThreadGroup parent, String n) {
      super(parent, n);
   }
   public void uncaughtException(Thread t, Throwable e) {
      System.out.println(t + " has unhandled exception:" + e);
   }
}
```

Let us compile and run the above program, this will produce the following result:

```
Starting Thread-0...
Starting Thread-1...
Thread-0 executing...
Thread-1 executing...
Thread-0 interrupted: java.lang.InterruptedException: sleep interrupted
Thread[Thread-0,5,ParentThreadGroup] has unhandled exception: java.lang.RuntimeException: sleep interrupted
Thread-1 interrupted: java.lang.InterruptedException: sleep interrupted
```

Thread[Thread-1,5,ChildThreadGroup] has unhandled exception: java.lang.RuntimeException

Benefits of exception handling:

Separating Error-Handling Code from "Regular" Code

- Propagating Errors Up the Call Stack
- Grouping and Differentiating Error Types

6.2 Termination or resumptive models:

The Java exception handling facility supports the termination model. In the termination model, when a method encounters an exception, further processing in that method is terminated and control is transferred to the nearest exception handler that can handle the type of exception encountered. It's important to note that this doesn't necessarily mean the entire program is terminated. It depends on what action the exception handler performs and where the exception handler is in the call chain. For example, the program in Listing 1 continues even after an exception has been thrown.

Listing 1 Sample exception processing in which part of the program continues to execute.

```
class simple_exception extends Exception{
 int Value;
 public simple_exception()
 {
  Value = -1;
 }
 public simple_exception (String Desc)
 {
  super(Desc);
 }}
public class example1
{
 example1(){}
 public boolean a() throws simple_exception
 {
  if(exceptional_condition){
   simple_exception SimpleException = new simple_exception();
   throw SimpleException;
  }
  // more processing for a()
  return(true);
 }

 public void b()
```

```
{
 boolean Result;
 try{
  Result = a();
  // more processing
  ...

 }
 catch(simple_exception SomeException)
 {
 // execute exception handling strategy
 }
 finally{
 // regardless to what happens the code in this
 // finally block is executed...
 // more recovery & cleanup code required
 }
}
public void c()
{
 // important processing
 }
public static void main(String[] Args)
 {
  ...
  example1 Example1 = new example1();
  Example1.b();
  Example1.c();
}}
```

In the program in Listing 1, the method a() encountered an exceptional condition and was unable to continue its processing. The method b() called a(). Because a() threw an exception, b() was also unable to finish its processing. However, b() contains an exception handler that knows how to deal with simple_exception objects. So when a() throws the exception, b()'s handler is activated. b()'s exception handler uses some strategy to cause the program to be stabilized. Ideally, b()'s exception handler will bring the software back into a consistent state. Another important component to notice in b() is the finally{} block. Even if the code in a method is interrupted abruptly by an exception, the system will attempt to implement the finally block.

In Example1, b()'s processing is interrupted because a() throws an exception. b() contains an exception handler that handles the problem encountered in a(). Although b()'s regular flow of control is interrupted, the exception handler executes, and after it completes, the finally{} block executes. Regardless of the interruptions that can happen in b(), the finally block will be attempted.

Notice in Listing 1 that main() calls b() and then calls c(). Although b() couldn't complete its processing (it was terminated), c() still executes. Although Listing 1 contains an oversimplification of exception handling, it demonstrates that the termination model doesn't necessarily result in termination of the entire program.

The program in Listing 1 demonstrates some of the basics of the Java exception handling feature:

- Methods may throw exceptions, as is the case with a() in Listing 1.
- Methods will contain exception handlers, as is the case with b() in Listing 1.
- The finally block will always be executed, even if the try or catch block throws an exception.
- When a method throws an exception, the flow of control is transferred to the nearest method that can handle the type of exception that was thrown in a program.
- Exceptions have types, and user-defined types are always derived from built-in exception types. The basics of the Java

exception handling feature has at least two important implications for the software architecture:

- The flow of control in the software architecture can be changed by the throw feature.
- The exception classes used to introduce new types, and each type has its own semantics or meaning.

The transfer of control from the problem area to someplace that knows how to bring the system into a consistent state, and the semantics of the exception thrown, together allow us to start to reach for the goal of fault tolerance. The *semantics* of the exception thrown describe what the exceptional condition is and suggest what should be done. The transfer of control takes us to code that implements exception strategy. The exception strategy is designed to make the software resilient to defects and system failures. In Java, the catch() mechanism either implements the exception strategy directly or creates objects and calls methods that implement the exception strategy:

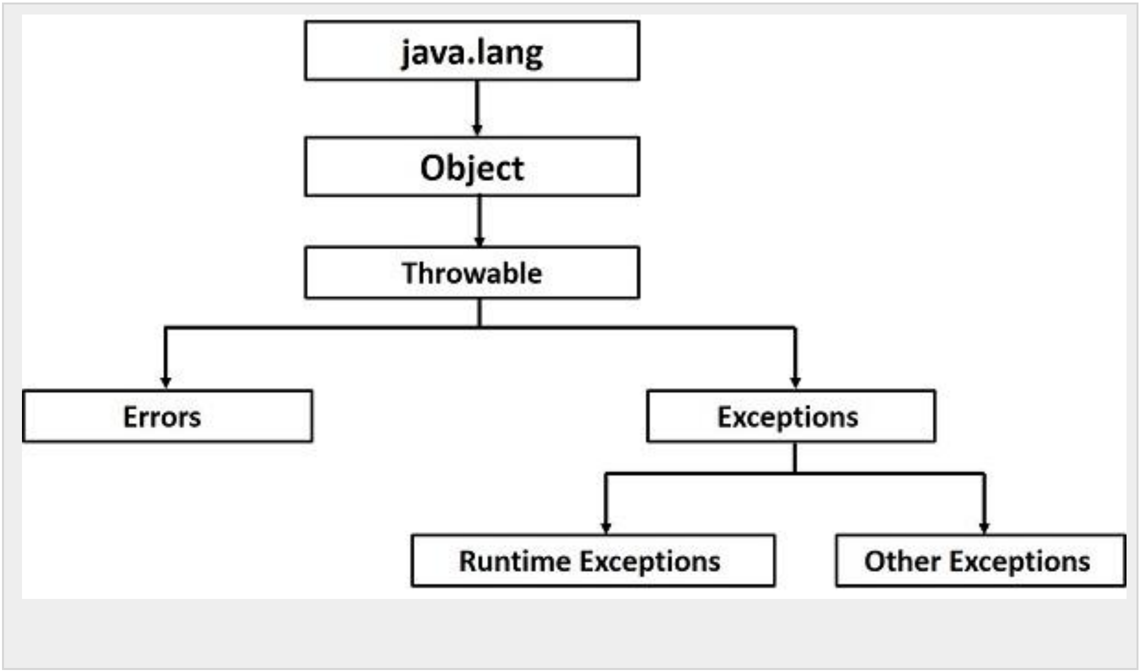
```
catch(some_exception){

// to write exception strategy

}
```

6.3 Exception Hierarchy:

All exception classes are subtypes of the java.lang. Exception class. The exception class is a subclass of the Throwable class and other than the exception class there is another subclass called Error which can be derived from the Throwable class.

Errors are abnormal conditions that happen in case of severe failures, these cannot be handled by the program itself. Errors are generated to indicate errors generated by the runtime environment.

For Example: JVM is out of memory. Normally, programs cannot recover from errors.The Exception class has two main subclasses: IOException class and RuntimeException Class.



6.4 Usage of try, catch, throw, throws and finally:

```
class TestException
{
 static void avg()
 {
  try
  {
   throw new ArithmeticException("demo of exceptions");
  }
  catch(ArithmeticException e)
  {
   System.out.println("Exception caught here");
  }
 }
Finally{
System.out.println(“ The execution of exception handling
completed and this block executes compulsory”);
system.Exit(1);
}
 public static void main(String args[])
 {
  avg();
 }
}
```

Java Built-in Exceptions:

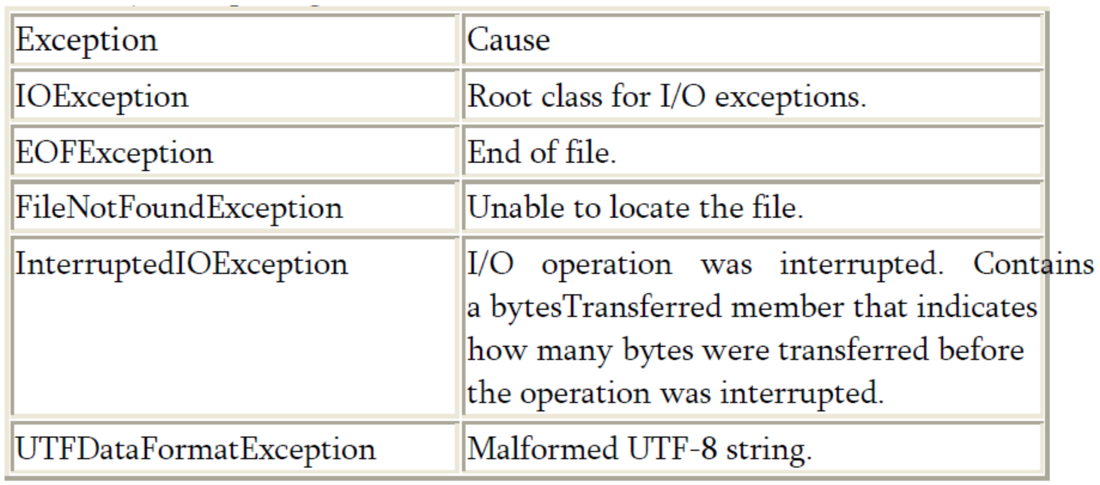
The various packages included in the Java Developers Kit throw different kinds of Exception and Error exceptions, as described below.

java.lang Exceptions

The java.lang package contains more core Java language. The exceptions sub classed from Runtime Exceptions cannot be declared in a method's throws clause. These Exceptions are considered normal and nearly any method can throw them in program execution.

java.io Exceptions

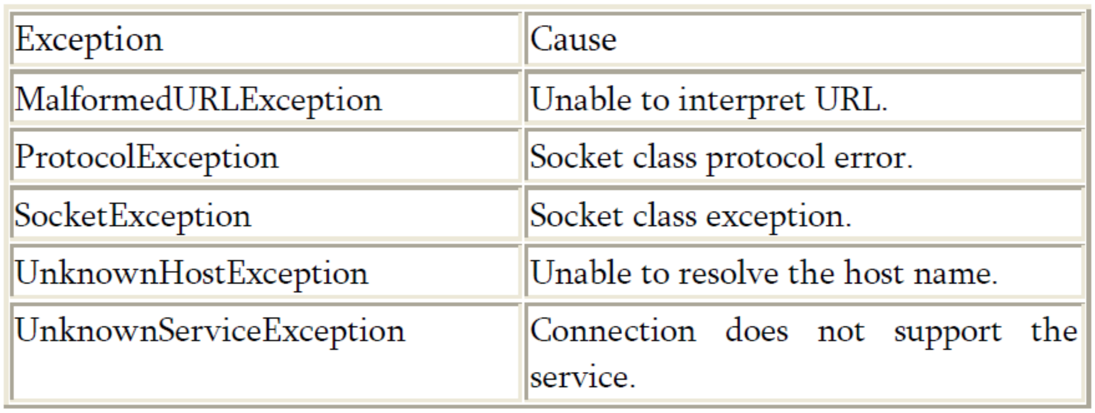
Any classes that work with I/O are good candidates to throw recoverable exceptions. For example, activities such as opening files or writing to files are likely to fail from time to time while execution. The classes of the java.io package do not throw errors at all.



Exception	Cause
IOException	Root class for I/O exceptions.
EOFException	End of file.
FileNotFoundException	Unable to locate the file.
InterruptedIOException	I/O operation was interrupted. Contains a bytesTransferred member that indicates how many bytes were transferred before the operation was interrupted.
UTFDataFormatException	Malformed UTF-8 string.

java.net Exceptions

The java.net package is used in network communication. Its classes most often throw exceptions to indicate connect failures and the like recoverable exceptions from the java.net package. The classes of this package do not throw errors at all any time of execution.



Exception	Cause
MalformedURLException	Unable to interpret URL.
ProtocolException	Socket class protocol error.
SocketException	Socket class exception.
UnknownHostException	Unable to resolve the host name.
UnknownServiceException	Connection does not support the service.

java.awt Exceptions

The AWT classes have members that can throw one error and one exception:

AWTException (exception in AWT)

AWTError (error in AWT)

java.util Exceptions

The classes of java.util throw the following exceptions:

EmptyStackException (that is no objects on stack)

NoSuchElementException (no more objects in collection)

An example of built-in exception is given below:

```
import java.io.* ;
import java.lang.Exception ;
```

```
public class DivideBy0
{
public static void main( String[ ] args )
{
int a = 2 ;
int b = 3 ;
int c = 5 ;
int d = 0 ;
int e = 1 ;
int g = 3 ;
try
{
System.out.println( a+"/"+b+" = "+div( a, b ) ) ;
System.out.println( c+"/"+d+" = "+div( c, d ) ) ;
System.out.println( e+"/"+f+" = "+div( e, g ) ) ;
}
catch( Exception except )
{
System.out.println( "Caught exception " + except.getMessage() ) ;
} }
static int div( int a, int b )
{
return (a/b) ;
} }
```

The output of this application is shown here:

```
2/3 = 0
Caught exception / by zero
```

The first call to div() works fine. The second call fails because of the divide-by-zero error. Even though the application did not specify it, an exception was thrown-and caught. So you can use arithmetic in your code without writing code that explicitly checks bounds.

Creating own exception sub classes:

User can create his exception handling class.

Example :

```
class MyException extends Exception
{
private int detail;
MyException(int a)
{
detail = a;
}
public String toString( )
{
return "MyException[" + detail + "]";
}
}
class ExceptionDemo
{
```

```
static void compute(int a) throws MyException
{
System.out.println("Called compute(" + a + ")");
if (a > 10) throw new MyException(a);
System.out.println("Normal exit");
}
public static void main(String args[ ])
{
try
{
compute(10);
compute(40);
}
catch (MyException e)
{
System.out.println("Caught an exception " + e);
 } } }
```

Chained Exceptions:

The following example shows how to handle chained exception using multiple catch blocks.

```
public class Main{

   public static void main (String args[])throws Exception {

      int n = 30, result = 0;
      try {
         result = n/0;
         System.out.println("The result is"+result);
      } catch(ArithmeticException ex) {
         System.out.println ("Arithmetic exception occoured: "+ex);
         try {
            throw new NumberFormatException();
         } catch(NumberFormatException ex1) {
            System.out.println ("Chained exception thrown manually :
"+ex1);
         }
      }
   }
}
```

Output:

The above code sample will produce the following result.

```
Arithmetic exception occoured :

java.lang.ArithmeticException: / by zero

Chained exception thrown manually :

java.lang.NumberFormatException
```

6.5 Three Recently Added Exception Features:

Beginning with JDK 7 version, three interesting and useful features have been added to the exception system. The first automates the process of releasing a resource, such as a file, when it is no longer needed. It is based on an expanded form of the try statement called *try-with-resources*. The second feature is called *multi-catch*, and the third is sometimes referred to as *final rethrow* or *more precise rethrow methods*. These two features are described here. The multi-catch feature allows two or more exceptions to be caught by the same catch clause. It is not uncommon for two or more exception handlers to use the same code sequence even though they respond to different exceptions and Instead of having to catch each exception type individually, we can use a single catch clause to handle all of the exceptions without code duplication. To use multi-catch, we need to separate each exception type in the catch clause with the OR operator. Each multi-catch parameter is implicitly final. Because each multi-catch parameter is implicitly final, it can’t be assigned a new value.

CHAPTER 7

MULTITHREADING

7.0 Fundamentals:

Multithreading is a feature in java programming language and it can be defined as a single program is divided into two or more subprograms, which can be executed at the same time and subprogram is called Thread. Multi-threading enables to write efficient programs that make the maximum use of the CPU, keeping the idle time to a minimum.

There is plenty of idle time for interactive and networked applications in java:

1) The transmission rate of data over a network channel is much slower than the rate at which the computer can process it.

2) Local file system resources can be read and written at very slower rate than can be processed by the CPU.

3) User input is much slower than the computer.

The java Thread Model:

The **Java** run-time system depends on the **threads** for many things, and all the class libraries are designed with multithreading in mind. The value of a multithreaded environment can be understood to its counterpart. Single-threaded systems use an approach called an event loop with polling.

The Java run-time system depends on the threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses the threads to enable the entire environment to be asynchronous way. This helps to reduce inefficiency by preventing the waste of CPU cycles.

Single-threaded systems uses an approach called an *event loop* with *polling*. In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next step. Once this polling technique returns with a single that network file is ready to be read, then the event loop dispatches the control to the appropriate event handler. Until this event handler returns, nothing else can happen in the program. This wastes the CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed. In general, in a single-threaded environment, when a thread *blocks* (i.e., suspends execution) because it is waiting for some resource, the entire program stops running.

The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping the other parts of your program. For example, the idle time created when a thread reads the data from a network or waits for the user input can be utilized elsewhere. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All the other threads continue to run.

As most readers know, over past few years, multi-core systems have become commonplace. Of course, single-core systems are still in widespread use. It is important to understand that the Java's multithreading features work in both types of systems. In a single-core system, concurrently executing threads share the CPU, with each thread receiving a slice of CPU time. Therefore, in a single-core system, two or more threads do not actually run at the same time, but idle CPU time utilized. However, in a multi-core system, it is possible for two or more threads to actually execute simultaneously. In many cases, this can further improve the program efficiency and increase the speed for certain operations.

A thread can be running. It can be ready to run as soon as it gets CPU time. A running thread can be suspended, which temporarily halts its activity. A suspended thread can then be resumed, allowing it to pick up where it left off. A thread can be blocked when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, then the thread cannot be resumed.

Messaging System

After you divide your program into separate threads, you need to define how they will communicate with one another as instructions.

When programming with some other languages, we must depend on the operating system to establish communication between the threads. This, of course, adds overhead in execution. By contrast, Java provides a clean, low-cost way for two or more threads to talk to one another, via calls to predefined methods that all the objects have. Java's messaging system allows a thread to enter a synchronized method on an object, and then waits there until some other threads explicitly notifies it to come out of the execution.

7.1 Java Thread Class and Runnable Interface

Java's multithreading system is built upon the Thread class, its methods, and its companion interfaces. Thread encapsulates a thread of execution. Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the Thread instance that spawned it.

7.1.1 The main thread:

When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins execution. The main thread is important for two reasons:

• It is the thread from which other "child" threads will be spawned.

• It must be the last thread to finish execution. When the main thread stops, your program terminates.

Although the main thread is created automatically whenever program is started, it can be controlled through a Thread object. To do so, we must obtain a reference to it by calling the method currentThread(), which is a public static member of Thread. The general form is shown below:

```
static Thread currentThread( )
```

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

Let's begin by reviewing the following example:

```
// Controlling the main Thread.

class CurrentThreadDemo {

public static void main(String args[]) {

Thread t = Thread.currentThread();

System.out.println("Current thread: " + t);

// change the name of the thread

t.setName("My First Thread");
System.out.println("After name change: " + t);

try {

for(int n = 5; n > 0; n—) {

System.out.println(n);

Thread.sleep(2000);

}

} catch (InterruptedException e) {

System.out.println("Main thread interrupted");

}

}

}
```

In this program, a reference to the current thread (the main thread, in this case) is obtained by calling currentThread( ), and this reference is stored in the local variable t. Next, the program displays information about the thread. The program then calls setName( ) to change the internal name of the thread. The Information about the thread is then redisplayed in the console while execution. Next, a loop counts down from five, pausing one second between each line. The pause is accomplished by the sleep() method. The argument to sleep() specifies the delay period in milliseconds.

The output generated by above program as given below:

```
Current thread: Thread [main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

7.1.2 Creating Thread:

To create a new thread a program can do the following:

1) By extending the Thread class, or

2) And implementing the Runnable interface

Thread class encapsulates thread execution. The whole Java multithreading environment is based on the Thread class. The following are some of the methods which will be used to develop a thread.

Start: a thread by calling start its run method

Sleep: suspend a thread for a period of time

Run: entry-point for a thread

Join: wait for a thread to terminate

isAlive: determine if a thread is still running

getPriority: obtain a thread’s priority

getName: obtain a thread’s name

To create a new thread by implementing the Runnable interface :

```
class NewThread implements Runnable
{
Thread t;
NewThread()
{
t = new Thread(this, "Demo Thread creation");
System.out.println("Child thread: " + t);
t.start();
}
public void run( )
{
try {
for (int i = 3; i > 0; i--)
{
System.out.println("Child Thread: " + i);
Thread.sleep(300);
} }
catch (InterruptedException e)
{
System.out.println("Child interrupted thread.");
}
System.out.println("Exiting child thread.");
} }
```

```
class ThreadDemo
{
public static void main(String args[ ])
{
new NewThread();
try
{
for (int i = 5; i > 0; i--)
{
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
}
catch (InterruptedException e)
{
System.out.println("Main thread is interrupted.");
}
System.out.println("Main thread exiting.");
} }
```

To create a new thread by extending the thread class :

```
class NewThread extends Thread
{
NewThread( )
{
super("Demo Thread");
```

```
System.out.println("Child thread: " + this);
start( );
}
public void run( )
{
Try
{
for (int i = 5; i > 0; i--)
{
System.out.println("Child Thread: " + i);
Thread.sleep(300);
} }
catch (InterruptedException e)
{
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
} }
class ExtendThread
{
public static void main(String args[ ])
{
new NewThread();
try
{
```

```
for (int i = 5; i > 0; i--)

{
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
} }
catch (InterruptedException e)
{
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
} }
```

7.2 Synchronizing Threads:

When two or more threads need access to a shared resource, they need a way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization.* Java implementation of synchronization:

1) Classes can define so-called synchronized methods

2) Each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called

3) Once a thread is inside a synchronized method, no other thread can call any other Synchronized method on the same object.

Example :

```
class CallYou
```

```
{
void call(String msg)
{
System.out.print("[" + msg);
try
{
Thread.sleep(2000);
}
catch (InterruptedException e)
{
System.out.println("Interrupted thread");
}
System.out.println("]");
}
}
class Caller implements Runnable
{
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s) {
target = targ;
msg = s;
t = new Thread(this);
t.
```

```
start( );
}
public void run( )
{
synchronized(target)
{
target.call(msg);
}
}
}
class Synch1
{
public static void main(String args[ ])
{
Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, "Synchronized");
Caller ob3 = new Caller(target, "World");
try
{
ob1.t.join();
ob2.t.join();
ob3.t.join();
}
catch(InterruptedException e)
```

```
{
System.out.println("Interrupted");
} } }
```

Here, the call( ) method is not modified by synchronized. Instead, the synchronized statement is used inside Caller's run( ) method. This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.

7.3 Thread Life Cycle:

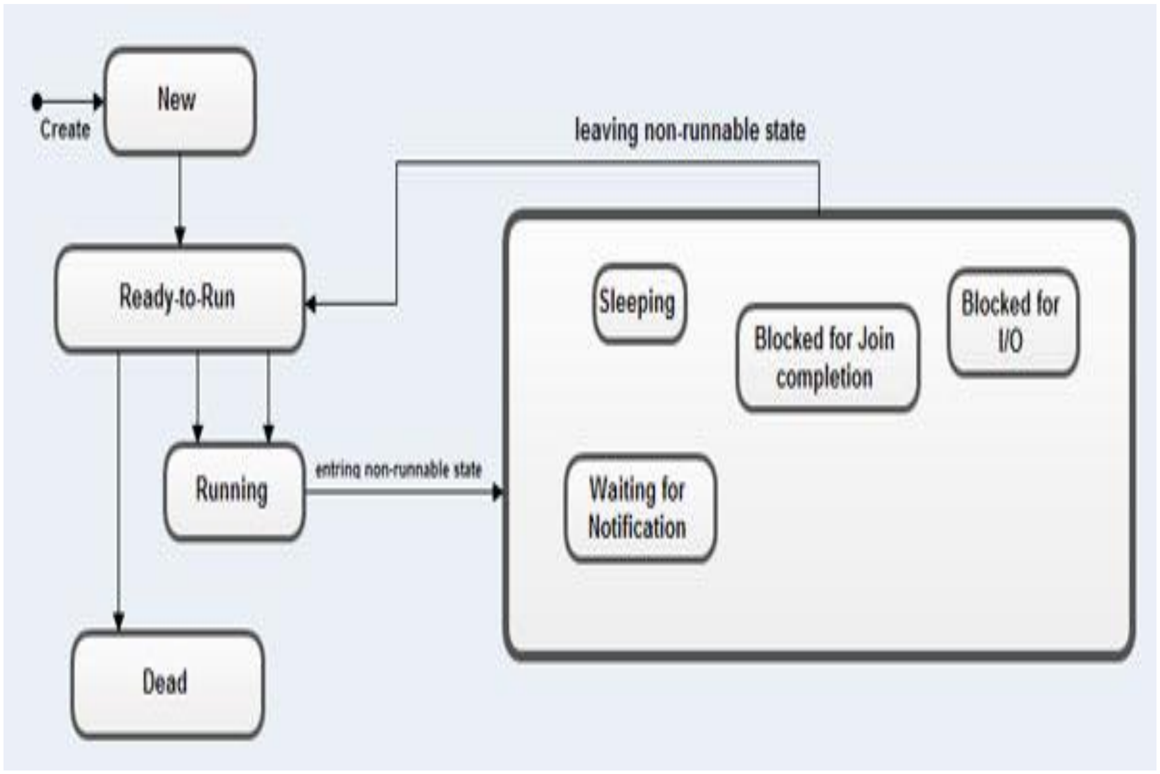
Thread can be defined as a sequence of instructions which can be run independently. Life cycle of a Thread has 5 states.

1. Born State
2. Runnable state.
3. Running State.
4. Blocked State.
5. Dead State

- Born state – After the creations of Thread instance the thread is in the state but before the start() method invocation. At this point, the thread is considered not alive.
- Runnable (Ready-to-run) state – A thread start its life from Runnable state. A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back

from blocked state also. On this state a thread is waiting for a turn on the processor.

- Running state – A thread is in running state that means the thread is currently executing. There are several ways to enter in Runnable state but there is only one way to enter in
- Running state: the scheduler select a thread from runnable pool.
- Dead state – A thread can be considered dead when its run() method completes. If any thread comes on this state that means it cannot ever run again.
- Blocked - A thread can enter in this state because of waiting the resources that are hold by another thread.


New
Create
leaving non-runnable state
Ready-to-Run
Sleeping
Blocked for Join completion
Blocked for I/O
Running
entring non-runnable state
Waiting for Notification
Dead

7.4 Runnable interface:

A **Runnable** is basically a type of class (**Runnable** is an **Interface**) that can be put into a thread, describing what the thread is supposed to do. The **Runnable Interface** requires of the class to implement the method run() like so:

```
public class MyRunnableTask implements Runnable
 {
 public void run()
 {
 // do write code here
 }}
```

7.5 Creating multiple threads:

The following example to illustrate creation multiple threads.

```
class NewThread implements Runnable {

  String name; // thread name

  Thread t;

  NewThread(String threadname) {

    name = threadname;

    t = new Thread(this, name);

    System.out.println("New thread name: " + t);

    t.start(); // Start the thread execution

  }

 public void run() {

    try {

      for (int i = 5; i > 0; i--) {

        System.out.println(name + ": " + i);

        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println(name + "Interrupted");
    }
    System.out.println(name + " exiting.");
  }
}
class MultiThreadDemo {
  public static void main(String args[]) {
    new NewThread("One"); // start threads
    new NewThread("Two");
    new NewThread("Three");

    try {
      Thread.sleep(10000);
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }
    System.out.println("Main thread exiting.");
  }
}
```

7.6 Thread priorities:

Each thread has a priority. Priorities will be represented by a number between 1 and 10. In most of the situations, thread priority schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

public static int MIN_PRIORITY

public static int NORM_PRIORITY

public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

An Example for priority of a Thread:

```
class TestMultiPriority1 extends Thread{

 public void run(){

  System.out.println("running thread name
is:"+Thread.currentThread().getName());

   System.out.println("running thread priority
is:"+Thread.currentThread().getPriority());   }

 public static void main(String args[]){

  TestMultiPriority1 m1=new TestMultiPriority1();

  TestMultiPriority1 m2=new TestMultiPriority1();

  m1.setPriority(Thread.MIN_PRIORITY);

  m2.setPriority(Thread.MAX_PRIORITY);

  m1.start();

  m2.start();

 }

}
```

7.7 Using isAlive() and join():

Joining threads

Sometimes one thread needs to know when another thread is ending. In java, isAlive() and join() are two different methods to check whether a thread has finished its execution.

The isAlive() method returns true if the thread upon which it is called is still running otherwise it returns false.

final boolean isAlive()

But, join() method is used more commonly than isAlive(). This method waits until the thread on which it is called terminates.

final void join() throws InterruptedException

Using join() method, we tell our thread to wait until the specified thread completes its execution. There are overloaded versions of join() method, which allows us to specify time for which you want to wait for the specified thread to terminate.

final void join(long milliseconds) throws InterruptedException

Example of isAlive method

```
public class MyThread extends Thread

{
```

```
public void run()
{
        System.out.println("r1 ");
        try {
        Thread.sleep(500);
        }
        catch(InterruptedException ie) { }
        System.out.println("r2 ");
}
public static void main(String[] args)
{
        MyThread t1=new MyThread();
        MyThread t2=new MyThread();
        t1.start();
        t2.start();
        System.out.println(t1.isAlive());
```

```
                System.out.println(t2.isAlive());
        }
}
```

Output :

```
r1
true
true
r1
r2
r2
```

Example of thread without join() method

```
public class MyThread extends Thread
{
        public void run()
        {
                System.out.println("r1 ");
        try {
        Thread.sleep(500);
        }
        catch(InterruptedException ie){ }
        System.out.println("r2 ");
    }
    public static void main(String[] args)
    {
        MyThread t1=new MyThread();
        MyThread t2=new MyThread();
        t1.start();
        t2.start();
    }
}
```

Output :

```
r1
```

```
r1

r2

r2
```

In this above program two thread t1 and t2 are created. t1 starts first and after printing "r1" on console thread t1 goes to sleep for 500 ms. At the same time Thread t2 will start its process and print "r1" on console and then go into sleep for 500 ms. Thread t1 will wake up from sleep and print "r2" on console similarly thread t2 will wake up from sleep and print "r2" on console. So you will get output like r1 r1 r2 r2

Example of thread with join() method

```
public class MyThread extends Thread

{

        public void run()

        {

                System.out.println("r1 ");

                try {

                Thread.sleep(500);
        }catch(InterruptedException ie){ }

        System.out.println("r2 ");

    }

    public static void main(String[] args)

    {

        MyThread t1=new MyThread();

        MyThread t2=new MyThread();

        t1.start();


        try{

            t1.join();      //Waiting for t1 to finish

        }catch(InterruptedException ie){}


        t2.start();

    }

}
```

Output :

```
r1

r2

r1

r2
```

In this above program join() method on thread t1 ensures that t1 finishes it process before thread t2 starts.

Specifying time with join()

If in the above program, we specify time while using join() with t1, then t1 will execute for that time, and then t2 will join it.

```
t1.join(1500);
```

Doing so, initially t1 will execute for 1.5 seconds, after which t2 will join it.

7.8 Thread communication:

Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of Object class:

wait()

notify()

notifyAll()

1) wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

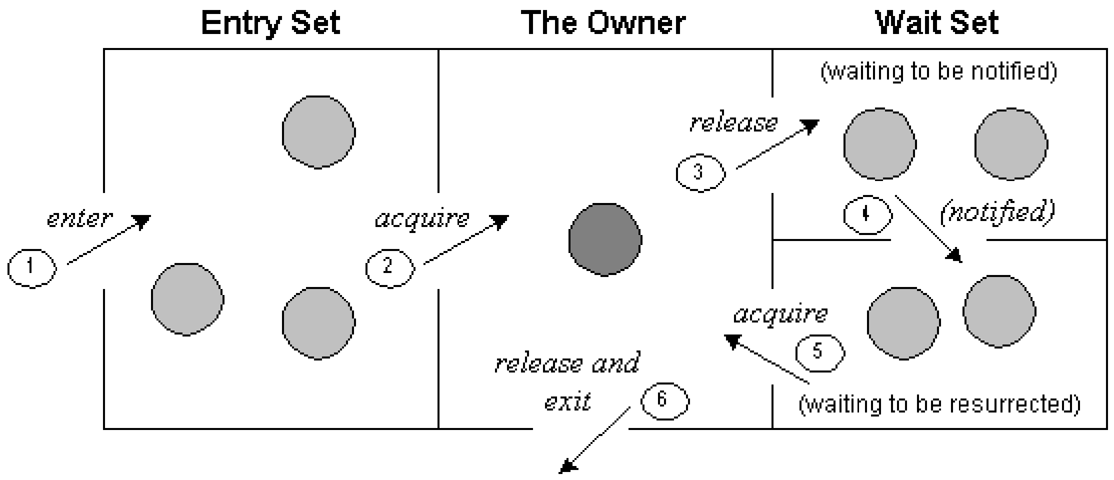
2) notify() method

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. Syntax:

public final void notify()

3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor. Syntax: public final void notifyAll()



Inter-Thread Communication

Obtaining a thread state:

Every thread moves through several states from its creation to its termination. The possible states of a thread are: NEW, RUNNABLE, WAITING, BLOCKED and TERMINATED. Immediately after the creation of a thread, it will be in the NEW state. After the start( ) method of the Thread class is executed, it will move to the RUNNABLE state. When the thread completes its execution, it will move to the TERMINATED stage.

If a thread is instructed to wait, it moves to the WAITING state. When the waiting is over, the thread once again moves to the RUNNABLE state. You can obtain the current state of a thread by calling the getState( ) method defined by Thread. It returns a value of type Thread. State, that indicates the state of the thread at the time at which the call was made. State is an enumeration defined by Thread. Given a Thread instance, you can use getState( ) to obtain the state of a thread.

7.9 Difference between multithreading and multitasking:

Multitasking is the ability to run several programs simultaneously, potentially by utilizing several processors, but predominantly, by time-sharing their resource requirements. An example is right on your desktop, where you may have a web browser, e-mail client, audio player, word processor, spreadsheet and who knows what else on the air at the same time. They can dance in and out of having the processor to themselves many times per second, because neither of them needs all of it for very long at a time. ***Multithreading***:

The ability to run several functions of a single program simultaneously, predominantly by utilizing several processors, but potentially, by time-sharing their resource requirements.

An example would be a web server, where the responses to all the incoming requests need much of the same program logic and state, but different handles on a few things (network socket, id of caller, whatever else). Sharing the greater bunch of the data pertaining to the program, but having dedicated copies of a small amount of private things, lets threads be spawned and destroyed very quickly, and permits an increase in available processing power to increase the number of requests answered without requiring an additional copies of the server program to be running. These are largely two sides of the same coin, the difference in vocabulary is mainly down to entire programs (processes) being a larger unit of stored state, with a correspondingly higher workload required to shift it around between processors and/or memory. Both of them require some O/S scheduling mechanism to keep track of which process/thread goes next, goes where and goes when, but the differences in the cost of manipulating processes and threads means that the best policy for one isn't necessarily good for the other. Hence, they get different names, and can be discussed as different things, within a context implied by the word chosen presently.

CHAPTER 8

I/O AND OTHER TOPICS

8.0 I/O basics:

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

Stream

A stream can be defined as a sequence of data. There are two kinds of Streams −

InPutStream − The InputStream is used to read data from a source.

OutPutStream − The OutputStream is used for writing data to a destination.

Streams

Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one.

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, FileInputStream and FileOutputStream. Following is an example which makes use of these two classes to copy an input file into an output file −

Example

```
import java.io.*;
public class CopyFile {

   public static void main(String args[]) throws IOException {
      FileInputStream in = null;
      FileOutputStream out = null;

      try {
         in = new FileInputStream("input.txt");
         out = new FileOutputStream("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Now let's have a file input.txt with the following content −

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

```
$javac CopyFile.java
$java CopyFile
```

Character Streams

Java Byte streams are used to perform input and output of 8-bit bytes, whereas Java Character streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, FileReader and FileWriter. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file −

Example

```
import java.io.*;
public class CopyFile {

   public static void main(String args[]) throws IOException {
      FileReader in = null;
      FileWriter out = null;

      try {
         in = new FileReader("input.txt");
         out = new FileWriter("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Now let's have a file input.txt with the following content −

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

```
$javac CopyFile.java
$java CopyFile
```

Standard Streams

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similarly, Java provides the following three standard streams −

Standard Input − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as System.in.

Standard Output − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as System.out.

Standard Error − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as System.err.

Following is a simple program, which creates InputStreamReader to read standard input stream until the user types a "q" −

Example

```
import java.io.*;
public class ReadConsole {

   public static void main(String args[]) throws IOException {
      InputStreamReader cin = null;

      try {
         cin = new InputStreamReader(System.in);
         System.out.println("Enter characters, 'q' to quit.");
         char c;
         do {
            c = (char) cin.read();
            System.out.print(c);
         } while(c != 'q');
      }finally {
         if (cin != null) {
            cin.close();
         }
      }
   }
}
```

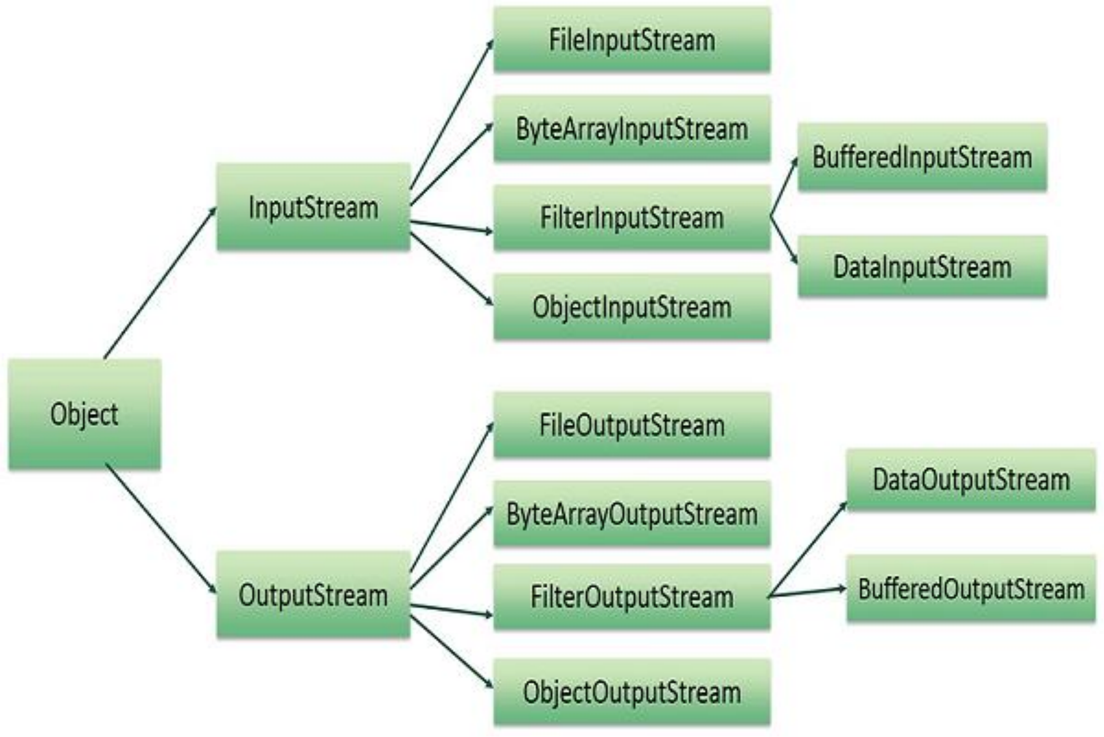
Let's keep the above code in ReadConsole.java file and try to compile and execute it as shown in the following program. This program continues to read and output the same character until we press 'q' −

```
$javac ReadConsole.java
$java ReadConsole
Enter characters, 'q' to quit.
1
1
e
e
q
q
```

8.1 Reading and Writing Files:

As described earlier, a stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



The two important streams are FileInputStream and FileOutputStream, which would be discussed in this tutorial.

FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword new and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file −

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows −

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

Once you have InputStream object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

There are other important input streams available, for more detail you can refer to the following.

ByteArrayInputStream

DataInputStream

FileOutputStream

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file −

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows −

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

Once you have OutputStream object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

There are other important output streams available, for more detail you can refer to the following.−

ByteArrayOutputStream

DataOutputStream

Example

Following is the example to demonstrate InputStream and OutputStream −

```
import java.io.*;
public class fileStreamTest {

   public static void main(String args[]) {

      try {
         byte bWrite [] = {11,21,3,40,5};
         OutputStream os = new FileOutputStream("test.txt");
         for(int x = 0; x < bWrite.length ; x++) {
            os.write( bWrite[x] );   // writes the bytes
         }
         os.close();

         InputStream is = new FileInputStream("test.txt");
         int size = is.available();

         for(int i = 0; i < size; i++) {
            System.out.print((char)is.read() + "  ");
         }
         is.close();
      }catch(IOException e) {
         System.out.print("Exception");
      }
   }
}
```

The above code would create file test.txt and would write given numbers in binary format. Same would be the output on the stdout screen.

Automatically closing a file:

Try with resources is a new feature in Java 7 which lets us write more elegant code by automatically closing resources like FileInputStream at the end of the try-block.

Old Try Catch Finally

Dealing with resources like InputStreams is painful when it comes to the try-catch-finally blocks. You need to declare the resources outside the try so that they are is accessible from finally, then you must initialize the variable to null and check for non-null when closing the resource in finally.

```
 File file = new File("input.txt");

  InputStream is = null;

  try {
    is = new FileInputStream(file);

    // do something with this input stream
    // ...
  }
  catch (FileNotFoundException ex) {
    System.err.println("Missing file " + file.getAbsolutePath());
  }
  finally {
    if (is != null) {
      is.close();
    }
  }
```

Java 7: Try with resources

With Java 7, you can create one or more “resources” in the try statement. A “resources” is something that implements the java.lang.AutoCloseable interface. This resource would be automatically closed and the end of the try block.

```
 File file = new File("input.txt");

   try (InputStream is = new FileInputStream(file)) {
     // do something with this input stream
     // ...
 }
  catch (FileNotFoundException ex) {
    System.err.println("Missing file " + file.getAbsolutePath());
   }
```

8.2 Random-access files:

Random access file is a special kind of file in Java which allows non-sequential or random access to any location in file. This means you don't need to start from 1st line if you want to read line number 10, you can directly go to line 10 and read. It's similar to array data structure, just like you can access any element in array by index you can read any content from file by using file pointer. A random access file actually behaves like a large array of bytes stored in file system and that's why its very useful for low latency applications which needs some kind of persistence e.g. in front office trading application and FIX Engine, you can use random access file to store FIX sequence numbers or all open orders. This will be handy when you recover from crash and you need to build your in memory cache to the state just before the crash. RandomAccessFile provides you ability to read and write into any random access file. When you read content from file, you start with current location of file pointer and pointer is moved forward past how many bytes are read. Similarly when you write data into random access file, it starts writing from current location of file pointer and then advances the file pointer past number of files written. Random access is achieved by setting file pointer to any arbitrary location using seek() method. You can also get current location by calling getFilePointer() method. There are two main ways to read and write data into RandomAccessFile, either you can use Channel e.g. SeekableByteChannel and ByteBuffer class from Java NIO, this allows you to read data from file to byte buffer or write data from byte buffer to random access file. Alternatively you can also use various read() and write() method from RandomAccessFile e.g readBoolean(), readInt(), readLine() or readUTF(). This is a two part Java IO tutorial, in this part we will learn how to read and write String from RandomAccess file in Java without using Java NIO channels and in next part we will learn how to read bytes using ByteBuffer and Channel API.

Following is our sample Java program, to demonstrate how to read or write String from a RandomAccessFile. The file name is "sample.store" which will be created in current directory, usually in your project directory if you are using Eclipse IDE. We have two utility method to read String from random access file and write string into file, both method takes an int location to demonstrate random read and random write operation. In order to write and read String, we will be using writeUTF() and readUTF() method, which reads String in modified UTF-8 format. Though, in this program, I have closed file in try block, you should always do that in a finally block with additional try block, or better use try-with-resource statement from Java 7. I will show you how you can do that in next part of this tutorial when we will learn reading and writing byte arrays using ByteBuffer and SeekableByteChannel class.

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

/**
* Java Program to read and write UTF String on
RandomAccessFile in Java.
*
* @author Javin Paul
*/
public class RandomAccessFileDemo{

   public static void main(String args[]) {

      String data = "KitKat (4.4 - 4.4.2)";
      writeToRandomAccessFile("sample.store", 100, data);
      System.out.println("String written into RandomAccessFile
from Java Program : " + data);

      String fromFile =
readFromRandomAccessFile("sample.store", 100);
      System.out.println("String read from RandomAccessFile in
Java : " + fromFile);
```

```
    }

    /*
     * Utility method to read from RandomAccessFile in Java
     */
    public static String readFromRandomAccessFile(String file, int
position) {
        String record = null;
        try {
            RandomAccessFile fileStore = new
RandomAccessFile(file, "rw");

            // moves file pointer to position specified
            fileStore.seek(position);

            // reading String from RandomAccessFile
            record = fileStore.readUTF();

            fileStore.close();

        } catch (IOException e) {
            e.printStackTrace();
        }
        return record;
    }

   /*
    * Utility method for writing into RandomAccessFile in Java
    */
    public static void writeToRandomAccessFile(String file, int
position, String record) {
        try {
            RandomAccessFile fileStore = new
RandomAccessFile(file, "rw");

            // moves file pointer to position specified
            fileStore.seek(position);

            // writing String to RandomAccessFile
            fileStore.writeUTF(record);

            fileStore.close();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Output:

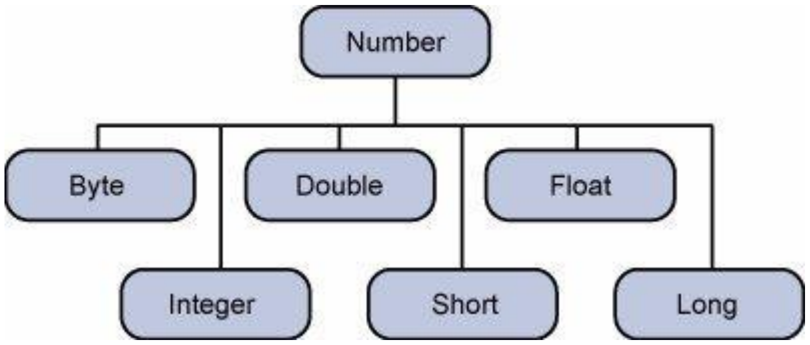
```
String written into RandomAccessFile from Java Program : KitKat
(4.4 - 4.4.2)
String read from RandomAccessFile in Java : KitKat (4.4 - 4.4.2)
```

8.3 Wrappers:

All the wrapper classes (Integer, Long, Byte, Double, Float, Short) are subclasses of the abstract class Number.

Number Classes

The object of the wrapper class contains or wraps its respective primitive data type. Converting primitive data types into object is called boxing, and this is taken care by the compiler. Therefore, while using a wrapper class you just need to pass the value of the primitive data type to the constructor of the Wrapper class.



And the Wrapper object will be converted back to a primitive data type, and this process is called unboxing. The Number class is part of the java.lang package.

Following is an example of boxing and unboxing −

Example

```
public class Test {
   public static void main(String args[]) {
      Integer x = 5; // boxes int to an Integer object
      x =  x + 10;   // unboxes the Integer to a int
      System.out.println(x);
   }
}
```

This will give the following result −

Output

```
15
```

When x is assigned an integer value, the compiler boxes the integer because x is integer object. Later, x is unboxed so that they can be added as an integer.

REFERENCES

1. The Complete Reference java by Herbert Schieldt
2. www.oracle.com
3. www.javatutorials.com
4. www.javatpoint.com
5. Java Wikipedia

PREVIOUS YEAR QUESTIONS

1. What is meant by responsibility? Explain it in detail.
2. Define class. Write and explain the hierarchy of classes.
3. Write short notes on “this” keyword and garbage collection in java.
4. Explain the different parameter passing techniques with example programs.
5. What is the use of “final” keyword? Explain with example program.
6. Give brief description about the abstract classes.
7. What is an interface? How can we implement multiple inheritance in java? Explain.
8. What is the use of CLASSPATH? Explain.
9. Draw and explain the life cycle of a thread.
10. Write short notes on java build in exceptions.
11. Explain the different layout managers in detail.
12. Differentiate between applet programming and application programming.
13. Draw and explain the life cycle of an applet program.
14. Explain in detail about the networking classes and interfaces.
15. Give brief description about the inetAddress.
16. What is the difference between message passing and a procedure call?
17. Briefly write about OOP principles.

18. Write a program to find the roots of quadratic equation.

19. What is type casting? What are the rules followed for type casting?

20. Explain the following:
    i. Super.
    ii. Static members of class.
    iii. Abstract methods Vs Concrete methods.

21. Write a java program to find date and time.

22. What is the significance of CLASSPATH environment variables in creating loosing a package?

23. Differentiate checked and unchecked exception.

24. What is thread? Explain thread life cycle.

25. Write the difference between AWT components and SWING components.

26. Write short notes on inner class and adapter class.

27. Write an applet program that display simple message " ALL THE BEST".

28. How do applets differs from application programs?

29. Write short notes on different types of applets.

30. Discuss briefly about the following: TCP,UDP & URL

31. Write a client-server application that takes the password as input and check whether it is correct.

32. The program should print the appropriate messages.

33. Briefly explain about abstract mechanism.

34. How to cope up with complexity? What are the mechanisms used to do so?

35. Explain briefly about bitwise operators.

36. Explain the structure of java program with an example.

37. What is multilevel inheritance? Explain with suitable example.

38. Explain how you can define constants in java. Explain with an example.

39. Write a java program to demonstrate implementing two interfaces by single class.

40. Discuss java.util package in detail.

41. Write a java program to implement runnable interface to create a thread.

42. Write short notes on Daemon threads.

43. Explain the following layout managers with a simple program.

    i. Card layout managers.

    ii. Grid bad layout managers.

44. What are mandatory attributes of applet tag? Explain them.

    i. Write a program which draws dashed line and dotted line using applet.

45. Explore java.net package.

46. Write about concept of responsibility in OOP.

47. Write about information hiding with respect to message passing.
48. What is an empty statement? Explain its usefulness.
49. Compare in terms of their functionalities, the following pairs while and do-while.
50. Can you declare abstract class members as final? Why?
51. Explain the following:
    i. Abstract methods.
    ii. Concrete methods.
52. Explain about classes of java.util package.
53. Write the difference between class and interface.
54. Explain the procedure to create user defined exceptions.
55. Why finally keyword is necessary in exception handling?
56. What are the functionalities supported by java related to drawing ellipses?
57. Explain about various networking classes & interfaces available in java.
58. What is the need of OOP paradigm?
59. Write about agents of communities.
60. How java is more secure than other languages?
61. What is data type? Explain data types in java.
62. What is method overloading? Explain with an example.

63. Explain the usage of final and super keywords with an example.

64. What is a package? How do create a package? Explain about access protection in packages.

65. What is an exception? Explain about exception handling mechanism in java.

66. Compare thread based and process based multitasking.

67. What is the task performed by layout manager? Explain different layout managers.

68. What is an applet? Explain applet life cycle.

69. Write the difference between applet and stand alone applications.

70. What is network programming? How are the different machines in a network addressed? Explain.

71. What is a part? What are the differences between port & socket? Explain with an example.

72. What is the difference between message passing and a procedure call?

73. Briefly write about OOP principles.

74. Write a program to find the roots of quadratic equation.

75. What is type casting? What are the rules followed for type casting?

76. Explain the following:

    i. Super.

ii. Static members of class.

iii. Abstract methods Vs Concrete methods.

77. Write a java program to find date and time.

78. What is the significance of CLASSPATH environment variables in creating loosing a package?

79. Differentiate checked and unchecked exception.

80. What is thread? Explain thread life cycle.

81. Write the difference between AWT components and SWING components.

82. Write short notes on inner class and adapter class.

83. Write an applet program that display simple message " ALL THE BEST".

84. Briefly explain about abstract mechanism.

85. How to cope up with complexity? What are the mechanisms used to do so?

86. Explain briefly about bitwise operators.

87. Explain the structure of java program with an example.

88. What is multilevel inheritance? Explain with suitable example.

89. Write a java program to demonstrate implementing two interfaces by single class.

90. What are the differences between private, static and final variables?

91. What is a package? How do we design a package?

92. How do we add a class or interface to a package?

93. What is the necessity of exception handling? Explain exception handling taking “Divide – by Zero” as an example.

94. What is dialog? Explain with an example.

95. Explain about graphics class.

96. What are the limitations of AWT?

97. Discuss model view architecture.

98. Define URL. What is the format of URL?

99. Explain the structure of java program with an example?

100. Create a base class with an abstract print ( ) method that is overridden in a derived class. The overridden version of the method prints the value of an int variable defined in the derived class. At the point of definition of this variable, give it a nonzero value. In the base-class constructor, call this method. In main ( ), create an object of the derived type, and then call its print ( ) method. Explain the results.

101. What is a package? How do create a package? Explain about access protection in packages.

102. Describe the life cycle of a thread with a neat sketch.

Dr. C .Shoba Bindu, received her Ph.D degree in Computer Science and Engineering from JNT University, Anantapur. She is currently working as an Associate Professor and Head of the Department in Computer Science & Engineering, JNTUA College of Engineering, JNT University, Anantapur. She has guided many external and internal projects and has good contributions in many of reputed journals. Her research interests are mobile and adhoc networks, network security, data mining and cloud computing. She has around 16 years of experience in teaching and Research.

Mr. P. Dileep Kumar Reddy, M.Tech is currently working as a Lecturer in Department of Computer Science & Engineering, JNTUA College of Engineering, JNT University, Anantapur. He completed his B.Tech & M.Tech in JNTUA College of Engineering, Anantapur. His areas of specializations are cloud computing, Network Security. He is in teaching since 2010. He has presented papers at National and International Conferences and published articles in National & International journals.

Dr. K. Dhanasree received her Ph.D degree in Computer Science and Engineering from JNTU, Anantapur. She is currently working as a professor in dept of computer science, DRKIST, Bowrampet. She has 15 years of teaching experience. Her research interests include Data Mining, Database Security, and Network Security.