1 INTRODUCTION TO JAVA

1.1 History

In 1991, a group of Sun Microsystems engineers led by James Gosling decided to develop a language for consumer devices (cable boxes, etc.). They wanted the language to be small and use efficient code since these devices do not have powerful CPUs. They also wanted the language to be hardware independent since different manufacturers would use different CPUs. The project was code-named Green.

These conditions let them to decide to compile the code to an intermediate machine-like code for an imaginary CPU called a virtual machine. (Actually, there is a real CPU that implements this virtual CPU now.) This intermediate code (called bytecode) is completely hardware independent. Programs are run by an interpreter that converts the bytecode to the appropriate native machine code. Thus, once the interpreter has been ported to a computer, it can run any bytecoded program.

Sun uses UNIX for their computers, so the developers based their new language on C++. They picked C++ and not C because they wanted the language to be object-oriented. The original

name of the language was Oak. However, they soon discovered that there was already a programming language called Oak, so they changed the name to Java.
Java is guaranteed to be “Write Once, Run Anywhere”.

Java Program Structure

A file containing Java source code is considered a compilation unit. Such a compilation unit contains a set of classes and, optionally, a package definition to group related classes together. Classes contain data and method members that specify the state and behavior of the objects in your program.
Java programs come in two flavors:
1. Standalone applications that have no initial context such as a pre-existing main window Application programs are stand-alone programs that are written to carry out certain tasks on local computer such as solving equations, reading and writing files etc.
The application programs can be executed using two steps

1. Compile source code to generate Byte code using Javac compiler.
2. Execute the byte code program using Java interpreter.

2. Applets for WWW programming
An applet is a Java program that runs within a Java-compatible WWW browser or in an appletviewer. To execute your applet, the browser: Creates an instance of your applet Sends messages to your applet to automatically invoke predefined lifecycle methods
The major differences between applications and applets are:
Applets are not allowed to use file I/O and sockets (other than to the host platform). Applications do not have these restrictions.
An applet must be a subclass of the Java Applet class. Applications do not need to subclass any particular class.
Unlike applets, applications can have menus. Unlike applications, applets need to respond to predefined lifecycle messages from the WWW browser in which they're running.

1.2 Object Oriented Programming

Object-oriented programming takes advantage of our perception of world An object is an encapsulated completely-specified data aggregate containing attributes and behavior Data hiding protects the implementation from interference by other objects and defines approved interface. An object-oriented program is a growing and shrinking collection of objects that interact via. Messages You can send the same message to similar objects the target decides how to implement or respond to a message at run-time Objects with same characteristics are called instances of a class. Classes are organized into a tree or hierarchy. Two objects are similar if they have the same ancestor somewhere in the class hierarchy You can define new objects as they differ from existing objects Benefits of object-oriented programming include: reduced cognitive load (have less to think about and more natural paradigm) isolation of programmers (better team programming) less propagation of errors more adaptable/flexible programs faster development due to reuse of code. Key Object-Oriented Concepts
An object-oriented program is a collection of objects with same properties and behavior are instances of the same class.

Objects have two components: state and behavior (variables and methods)
An object may contain other objects. Instance variables in every object, class variables are like global variables shared by all instances of a class

1.3 JAVA Bytecode

Why understand bytecode?
Bytecode is computer object code that is processed by a program, usually referred to as virtual, rather than by the "real" computer machine, the hardware processor. The virtual machine converts each generalized machine instruction into a specific machine instruction or instructions that this computer's processor will understand.
Java code is written in .java file. This code contains one or more Java language attributes like Classes, Methods, Variable, Objects etc. Javac is used to compile this code and to generate .class file. Class file is also known as "byte code".

1.4 Java Virtual Machine(JVM)

JVM is the main component of Java architecture and it is the part of the JRE (Java Runtime Environment). It provides the cross platform functionality to java. This is a software process that converts the compiled Java byte code to machine code.
A Java virtual machine's main job is to load class files and execute the bytecodes they contain. As you can see in Figure 1-3, the Java virtual machine contains a class loader, which loads class files from both the program and the Java API. Only those class files from the Java API that are actually needed by a running program are loaded into the virtual machine. The bytecodes are executed in an execution engine.

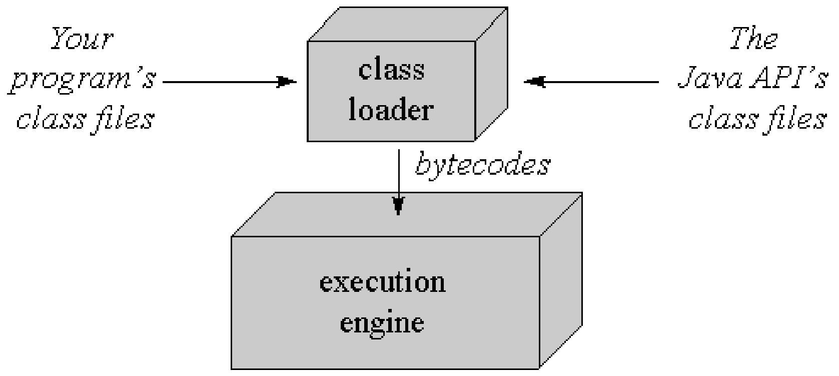


Figure . A diagram of basic block the Java virtual machine.

The execution engine is one part of the virtual machine that can vary in different implementations. On a Java virtual machine implemented in software, the simplest kind of execution engine just interprets the bytecodes one at a time.

1.5 JAVA Architecture

The Architecture
Java's architecture arises out of four distinct but interrelated technologies:

- The Java programming language
- The Java class file format

- The Java Application Programming Interface
- The Java virtual machine

When you write and run a Java program, you are tapping the power of these four technologies. You express the program in source files written in the Java programming language, compile the source to Java class files, and run the class files on a Java virtual machine.

When you write your program, you access system resources (such as I/O, for example) by calling methods in the classes that implement the Java Application Programming Interface, or Java API. As your program runs, it fulfills your program's Java API calls by invoking methods in class files that implement the Java API. You can see the relationship between these four parts inFigure 1-1.

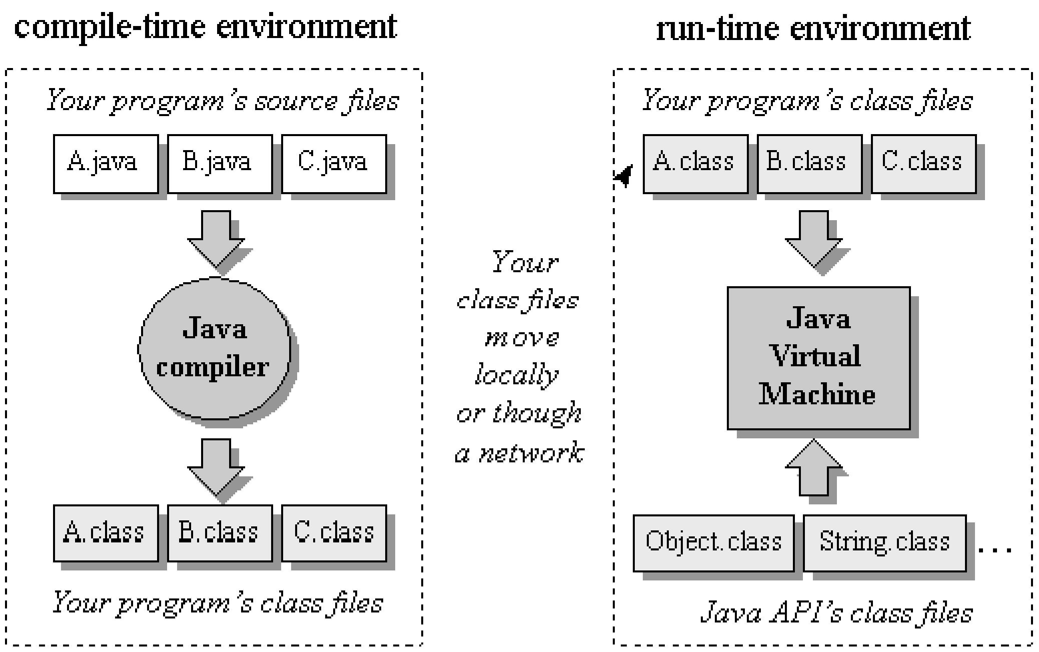


Figure 1-1. The Java programming environment.

Together, the Java virtual machine and Java API form a "platform" for which all Java programs are compiled. In addition to being called the Java runtime system, the combination of the Java virtual machine and Java API is called the Java Platform (or, starting with version 1.2, the Java 2 Platform). Java programs can run on many different kinds of computers because the Java Platform can itself be implemented in software. As you can see in Figure 1- 2, a Java program can run anywhere the Java Platform is present.

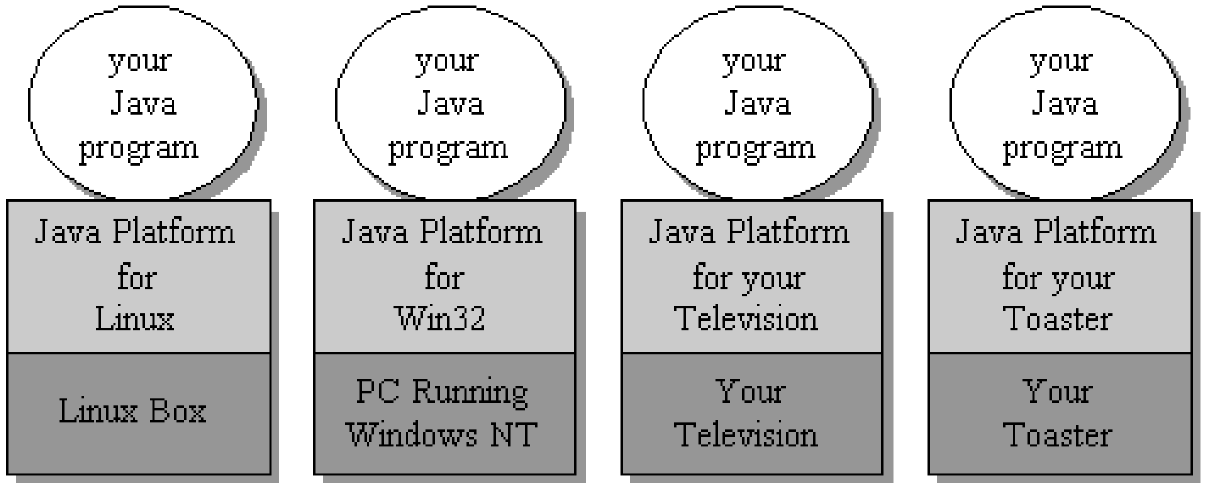


Figure : Java programs run on top of the Java Platform.

1.6 Properties or Buzzwords of JAVA

The JAVA Buzzwords

Java has many features which make it better of all other OOPS based programming languages, these are also known as buzzwords of java. These are as follow:

- Simple
- Robust
- Object Oriented
- Secure
- Architecture Neutral
- Portable
- Interpreted
- Multi-threaded
- Dynamic

1. Simple: Java is so designed that it is easy to learn. Syntax of java is kept very near to C++. It does not mean that it has no complexity. It simply means, If a person have knowledge of C++ then it is easier for him.

2. Robust: To develop a multi-platform envoronment it must be ensured that program must be relibly execute on any platform.

Java provides compile time as well as run time error checking. During compile time syntax error in programs are checked and at run time exceptions are trapped.

Java memory management does not allow creation of pointers and has automatic garbage collection once the memory is not required.

3. Object Oriented: In java everything is an object. Program runs around properties and behavior of objects. Java uses objects for Software design.

4. Secure: Java is designed to work run on distributed as well as network environment where security measures should be high. Java supports creation of application which does not invade form outside.
–>Java does not create .exe file which are most prone to virus attack.
–>Java Supports SAND Box security so network code can’t see machine hardware.
5. Architecture Neutral: Write once; run anywhere, any time, forever.Java was designed to run on any machine architecture in network. Java compiler generates byte code instruction which does not depend on any machine architecture. This machine gets interpreted on any machine and translated into native machine code.
6. Portable: Java has no implementation dependence like C & C++. The size of primitive data type is same on any platform so the program. There is no hardware or software incompatibility across different architecture.
7. Interpreted: Java byte codes are interpreted before running on system, So the speed is little bit slower. Sun Micro System has recently developed java chips which makes it little bit faster. It still works to improve its speed.
8. Multi threaded: There are two ways to achieve multitasking i.e. process based and thread based. In java multiple threads are synchronized at a time. So it supports multiprogramming. Java has many high level libraries to support be thread safe, it means there is no conflict to multiple concurrent threads of execution.
Note:- Thread is running instance of a program.
9. Dynamic: Java supports run time data binding and linking. Classes are linked only when it needed. However the java compiler is strict in its static checking during compile time.

1.7 Java Basic Syntax

About Java programs, it is very important to keep in mind the following points.

- Case Sensitivity: Java is case sensitive which means identifier Hello and hello would have different meaning in Java.
- Class Names: For all class names the first letter should be in Upper Case.If several words are used to form a name of the class each inner words first letter should be in Upper Case. Example class MyFirstJavaClass
- Method Names: All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.
  Example public void myMethodName()
- Program File Name: Name of the program file should exactly match the class name.When saving the file you should save it using the class name (Remember java is case sensitive) and append '.java' to the end of the name. (if the file name and the class name do not match your program will not compile).

Example: Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as'MyFirstJavaProgram.java' public static void main(String args[]) - java program processing starts from the main() method which is a mandatory part of every java program.

- Java is a free from language
- All statements in java end must with semicolon (;)

1.8 Comparison of JAVA and C++

Java is built upon C and C++ language it derives its syntax from C and the Object Oriented fractures from C++.

JAVA	C++
Java is platform Independent	C++ is platform dependent.
Strongly influenced by C syntax, with Object-Oriented features added.	Strongly influenced by C++/C syntax.
Java does not support multiple inheritance and operator overloading	C++ support multiple inheritance and operator overloading
Java has its own automatic garbage collection mechanism.	C++ due to absence of automatic garbage collection mechanism problems like memory leaks and much development time is been taken for coding
Re-declaration of static data members outside the class are not required in Java	In C++ you have to re-declare static data members outside the class.
Java does not support Enums, Structures or Unions but supports classes	C++ support Enums, Structures or Unions but supports classes
Java does not required such things.	Scope resolution operator (::) required in C++

1.9 Comparison of JAVA Application and JAVA Applets

APPLET	STAND ALONE APPLICATION
An applet is an application designed to be transmitted over the Internet and executedby a Java-compatible Web browser.	An *application* is a program that runs on your computer, under the operating system of that computer.
An applet is an *intelligent program*, not just an animation or media file.	An application is just a program.
An applet is a program that can react to user input and dynamically change	An application can't change dynamically or react on user input.
Applets are normally embedded within HTML documents.	An applications are not embedded with HTML
Applet don't need to write main() method.	In each application we must have to write main() method.
Applet provides windowing, networking support.	An application doesn't provide these facilities.
Method of running applet is different than an application.	Method of running application is different than an applet.
Applet doesn't need java compiler to compile.	Applications need java compiler to compile.

Applets are faster than applications.	Applications are slower than applets.
Creating java applets is more difficult because we have to write it in java language.	Creating an application is easy.
Applets depend on a Java-capable browser in order to run.	Java applications don't require a browser to run

1.10 General Structure of JAVA program

Optional section

1. Document section
2. Package statement
3. Import statement
4. Interface statement

Compulsory section

5. Class definition
6. Main method class

A java program may contain one or more section

Main method and class is essential section(5 and 6 compulsory) and other are optional

1. Document section:

Document section contain a set of comments lines giving the name of the program, the author and others details which the programmer would like to refer to at a later stage

2. Package Statements:

The first statement allowed in JAVA is a Package statement. It declare a package name and informs the compiler that classes define here belong to these package

3. Import statement:

This is similar to the #include statement in c.

Syntax: import packagename.classname;

Example: import java.io.*;

The above statement instruct the interpreter to load the IO class contain in the package java

4. Interface statement:

An interface is like class but include a group of method declaration. This is also an optional section and it is used also when we wish to implement the multiple inheritance features in program.

Compulsory section

5. Class definition:

A java program may contain multiple class definition. Classes are primary and essential elements of java.

6. Main method class:

Since every java stand alone program required a main(). As it is starting point this is the essential part of java program. All java application begin execution by calling main(). A simple java program contain only this part.

1.11 Getting started with Programming in JAVA

Java programs: a stand-alone Java application and an applet that you can view in either in the appletviewer (part of the JDK) or in a Java-capable browser. Although both these programs are extremely simple, they will give you an idea of what a Java program looks like andhow to compile and run it.

1.12 Getting the Software

In order to write Java programs, you will, of course, need a Java development environment. At the time this book is being written, Sun’s Java Development Kit provides everything you need to start writing Java programs. The JDK is available for Sun SPARC systems running Solaris 2.2 or higher and for Windows NT,Windows 95 and higher versions.

1.13 Creating Your First Application

Your first application, HelloWorld, will simply display the greeting "Hello world!". To create this program, you will:

- **Create a source file**

A source file contains code, written in the Java programming language, that you and other programmers can understand. You can use any text editor to create and edit source files.

- **Compile the source file into HelloWorld .class file**

The Java programming language compiler (javac) takes your source file and translates its text into instructions that the Java virtual machine can understand. The instructions contained within this file are known as bytecodes.

- **Run the program**

The Java application launcher tool (java) uses the Java virtual machine to run your application.
Create a Source File
To create a source file, you have two options:

- You can save the file HelloWorld.java on your computer and avoid a lot of typing. Then, you can go straight to Compile the Source File into HelloWorld.class File.
- Or, you can use the following (longer) instructions.

First, start your editor. You can launch the Notepad editor from the Start menu by selecting Programs > Accessories > Notepad. In a new document, type in the following code:

```
/* The HelloWorld class implements an application that
 * simply prints "Hello World!" to standard output.
 */
class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello World!"); // Display the string.
   }
}
```

1.14 Introduction to JDK Tools

Using the command-line Java Development Kit (JDK) may be the best way to keep up with the very latest improvements from Sun/JavaSoft. This is not the fastest compiler available by any means, the compiler is written in Java and interpreted at compile time, making it a sensible bootstrapping solution, but not necessarily optimal for speed of development. Nonetheless, using Sun's JDK (or Java SDK), the commands are javac to compile and java to run your program. For

For example:

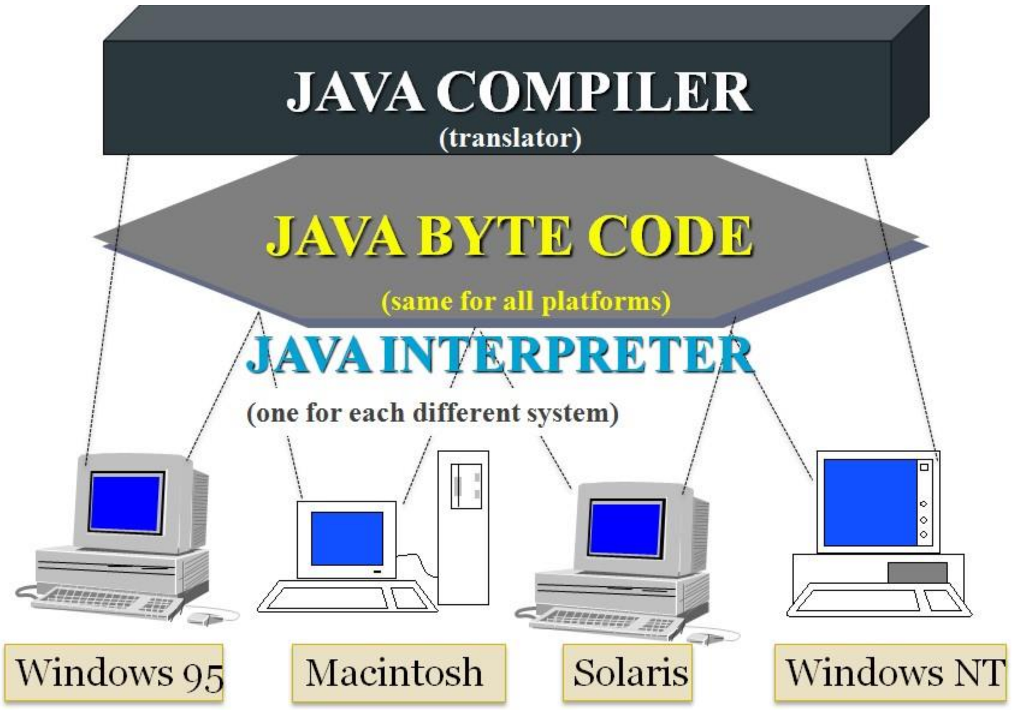
```
C:\java>javac HelloWorld.java
C:\java>java HelloWorld
Hello, World
```

1.15 JDK Tools

1.15.1 Java Compiler

A Java compiler is a compiler for the programming language Java. The most common form of output from a Java compiler is Java class files containing platform-neutral Java bytecode, but there are also compilers that produce optimized native machine code for a particular hardware/operating system combination.

JDK tools file for compiler is javac



you can create java programs using any text editor. Te file you create should have the extension "<u>.java</u>". every file that you create can have maximum of 1 publlic class definition The source code is converted into byte code by javac compiler. The javac compiler converts .java file into .class file, which contains byte code.

Syntax: javac [options] filenameOptions:

-classpath: overwrite the default classpath environment variable that contains the standardpath for library classes.
-d: specifies the directory for placing the resulted byte code.
-nowarn: turns off the warning messages.
-o: turns optimization on causing all static, final and private method to be placed inline. Although this result is faster in execution. The .class file becomes larger.
-verbose: generates the message, what compiler is doing.

1.15.2 Java Interpreter

An Interpreter directly executes instructions written in a programming or scripting language without previously converting them to an object code or machine code. Java programs are first compiled to an intermediate form, then interpreted by the interpreter

JDK tools file for compiler is java

the java interpreter is used to execute the compiled java application. The byte code which isthe result of compilation is interpreted so that it can be executed.

Syntax: java [options] filenameOptions:

-help: it shows all the options.
-version: it displays the version of the jdk that is used to compile the source code
-v(verbose): it displays the classes as they are loaded.
-cs: it check to see if the source file is newer than classfile. If source file is newer then new version of the source file is compiled.
-prof: it generates the information to the file java prof.
-classpath: it looks for the classfiles in the specified directory.
-verbosegc: it prints the message each time garbage collection occurs.
-noclassgc: it disables the garbage collection class.
-verify: it verifies whether all classes are loaded.
-noverify: turns off class verification.

1.15.3 Java Debugger

The JDK includes a command-line-based debugger, jdb, and there are any number of IDEs that include their own debugging tools. If you've focused on one IDE, learn to use the debugger that it provides. If you're a command-line junkie like me, you may want to learn at least the basic operations of jdb.

JDK tools file for Java Dibugger is JDB:

java debugger is a command line driven, debugging tool for the java environment. You canuse the debugger to debug local and remote files.

Syntax: JDB [options] filenameOptions:

–host: [JDB –host hostname]

it specifies the debugger where the remote java program reside. Hostname is the name of remote computer or you can give the ip address.

–password: [JDB –password passwordname]

It specifies the password require to access remote java file. The files are issued by java interpreter using –debug option.

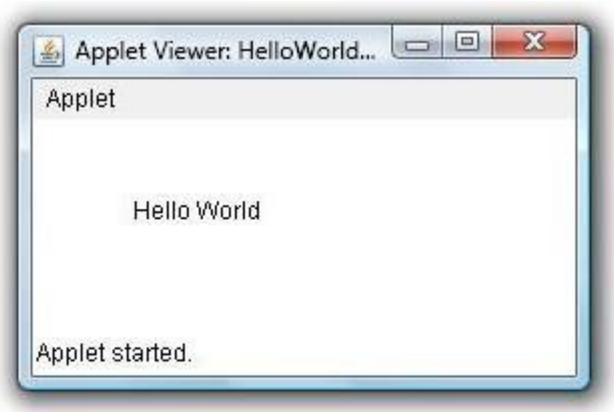
1.15.4 Applet Viewer

There is a program in the JDK known as Appletviewer, a kind of mini-browser. You need to give it the HTML file, just like a regular browser. Sun's AppletViewer (shown in figure) has an explicit reload button that actually reloads the applet. It has a View->Tag menu that lets you resize the window until the applet looks best, and then you can copy and paste the tag -- including the adjusted WIDTH and HEIGHT tags -- into a longer HTML document.

Following is Sun JDK AppletViewer

JDK file is appletviewer:

Aapplets are java programs that are emabaded in web pages. You can run applets using



a web browser. The applet viewer lets you run applets without the overhead of running aweb browser. You can test your applet using the appletviewer.

Syntax: appletviewer (url/path of filename)Options:

Restart(): restarts the applet using the current seconds.

Reload(): reload the applet with the changes in the .class file.

Stop():causing the stop method of the applet to be called and halts the applet

Save(): saves the serialized state of the applet

Clone(): duplicates the current applet to create another appletviewer instance.

Print():it causes the applet graphics to be sent to printer.

Tag(): it shows the html <applet> tag, that is used to run the current applet.

Close(): it closes the appletviewer window and terminates the applet (temporary).

Quit(): it closes the appletviewer window and terminates the applet (permanent).

1.15.5 Other JDK Tools files

These tools are the foundation of the JDK. They are the tools you use to create and build applications.

1.15.5.1 Javap:

If you do not have .java file but have byte code, you can retrieve the java file using disassambler. The java disassembler javap is used to retrieve source code from the byte code.Syntax: Javap <klist of classfile>

1.15.5.2 Javadoc:

Java doc is a document generator that creates html page documentation for the classes that you create. To use the java code, you have to embed the statements within /* */ or //. The java doc tool has been used by sun micro system for creating java documentation.
Syntax: javadoc <javafilename>
It create HTML file.

1.15.5.3 Javah:

The javah tool creates header files that let you extend your java code with c language.
Syntax: javah <classfilename>

1.15.5.4 Jar:

[java archive] jar utility of java is an archiving tool that combines multiple files into single archive files by using compression algorithm. Although JAR is a file archiving and compressionutility. Its main purpose is to compile files used by applet into single file for efficient loading by web browser. The loading is efficient because the browser has to establish a single HTTP connection with the web browser which reduces the working of browser and server.
Syntax: jar [options] <archive filename> <list of filename>

1.16 Summary:

Tool Name	Brief Description
Javac	The compiler for the Java programming language. the Java compiler, which converts source code into Java bytecode
Java	The loader for Java applications. This tool is an interpreter and can interpret the class files generated by the javac compiler. Now a single launcher is used for both development and deployment. The old deployment launcher, jre, no longer comes with Sun JDK, and instead it has been replaced by this new java loader.
Javadoc	API documentation generator. the documentation generator, which automatically generates documentation from source code comments
Apt	Annotation processing tool.
Appletviewer	Run and debug applets without a web browser.
Jar	Create and manage Java Archive (JAR) files. the archiver, which packages related class libraries into a single JAR file. To aggregate and compress multiple files into a singe JAR file.

Jdb	The Java Debugger.
Javah	C header and stub generator. Used to write native methods.
Javap	Class file disassemble
Extcheck	Utility to detect Jar conflicts

1.17 Exercise

1. What do you know about Java?
2. What are the supported platforms by Java Programming Language?
3. List any five features of Java?
4. Why is Java Architectural Neutral?
5. How java is object oriented?
6. What is bytecode?
7. How Java enabled High Performance?
8. Why Java is considered dynamic?
9. What is Java Virtual Machine and how it is considered in context of Java's platform independent feature?
10. List some Java keywords(unlike C, C++ keywords)?
11. What is the extension name of a Java bytecode file?
12. Explain How Jit Compiler Works
13. How Java is a truly Object Oriented Language? Explain.
14. Java Vs. C++.
15. Discuss Architecture Of Java. And Explain How Java Is Platform Independent.
16. Explain portability of Java.
17. State Difference between java interpreter and java compiler.
18. Write use of java debugger.
19. Explain java interpreter.
20. What is appletviewer?
21. List out jdk tools.
22. The output of the Java compiler is known as -----------------.
23. The .org part of a domain name stands for -------------- --.
24. The-----------statement is used to include another Java package in a Java source file.
25. A subclass can call a constructor method defined by its super class by use of the ------------------- keyword.
26. URL stands for----------------- --.

27 ------------------ is the protocol used to transmit hyper text over the Internet.

28. In Java, the AWT classes are contained in the ---------package.

29 is the java compiler.

30 command is used as the Java interpreter.

31. Java supports------------------- programming.

Answer

22. Byte code
23. Organization
24. import

25. Super
26. Uniform Resource Locator
27. HTTP
28. Java.awt
29. javac
30. Java
31. Multi threaded

2 Basics of JAVA

2 Basics of JAVA

Java has emerged as the language of choice among many educators for teaching introductory computer science. A clean, type-safe language, Java provides a garbage collected heap and a comprehensive exception-handling mechanism. Java Programming is a tightly integrated programming environment for live software construction in Java.

2.1 Program in Java

Your first application, HelloWorld, will simply display the greeting "Hello world!". To create this program, you will:
Create a source file
A source file contains code, written in the Java programming language, that you and other programmers can understand. You can use any text editor to create and edit source files.
Compile the source file into HelloWorld .class file
The Java programming language *compiler* (javac) takes your source file and translates its text into instructions that the Java virtual machine can understand. The instructions contained within this file are known as *bytecodes*.
Run the program
The Java application *launcher tool* (java) uses the Java virtual machine to run your application.
Create a Source File
To create a source file, you have two options:
You can save the file HelloWorld.java on your computer and avoid a lot of typing. Then, you can go straight to Compile the Source File into HelloWorld.class File.
Or, you can use the following (longer) instructions.

First, start your editor. You can launch the Notepad editor from the **Start** menu by selecting **Programs > Accessories > Notepad**. In a new document, type in the following code:

```
/**
 * The HelloWorld class implements an application that
 * simply prints "Hello World!" to standard output.
 */
class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello World!"); // Display the string.
   }
}
```

Output

```
C:\java> javac HelloWorld.java
C:\java> java  HelloWorld
Hello World!
```

<u>Basic Syntax of Java</u>

1. **Java file extension:** java source file save with .java extension.
2. **Java class file:** onece .java file is successfully compiled it create its .class file.
3. **Java file name and class name:** the name of java source file is same as name of main class file. in case of multiple classes in .java file the class is said to be main class if it contain main()
4. **The main method:** java source file contain main() as
   public static void main(String args[]){}
5. **Case sensitivity:** Java programming language is case sensitive as there is different meaning between **Main** and **main** in java.

2.2 Comments in Java

Comments are non executable lines to explain program and author details which make program understandable by programmer thought it is not created by him/her.
In java comments can be define by following way.

1. Single line comments
2. Multiline comments

Single line comments

Single line comment specify by **//** (double forward slash)
For Example:

Method 1: specify comment in separate line

```
// This is a single line comment
System.out.println("Jump2Learn : new way of learning ");
```

Method 2: Comment in same line

```
System.out.println("Jump2Learn : new way of learning "); // This is a single line comment
```

Multiline comments

Multiline comments specify by /* and*/ any text between /* and */ will be consider as non executable line.

For Example:

```
/* This program is illustration of multiline comments
 To print greetings */
System.out.println("Welcome to Jump2Learn");
```

2.3 JAVA Identifier

- Identifier means any legal names of classes, variables, methods , functions etc. when we define names for classes, variables, and methods are called identifiers.
- In Java, followings are rules to define valid identifirs:
    a. All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).
    b. After the first character, identifiers can have any combination of characters.
    c. A key word cannot be used as an identifier.
    d. Most importantly, identifiers are case sensitive.

    Examples of legal identifiers: **age, $charges, _point, __1_value.**

    Examples of illegal identifiers: **1a, -ve.**

2.4 Java Literals

Any constant value which can be assigned to the variable is called as literal/constant

1. **Integer Literals**
    - Most Commonly used type in typical program is any whole number value is an integer literal.
    - **For Example:** 1, 2, 3 and 25
    - Note: byte, short, long literals value is also type as same Integer literals
2. **Floating Point Literals**
    - Floating point number represents decimal values with fractional components.
    - **For Example:** 6.023E23, 3.1415
3. **Boolean Literals**
    - There are only two logical values that a Boolean value can have **true** or **false**
4. **Character Literals**
    - Characters in JAVA are indicates into the Unicode character set. Character literal is represent inside a pair of single quote (**' '**)
    - **For Example:** 'x','y'
5. **String Literals**
    - String literals in JAVA are specified like they are in most other languages by enclosing a sequence of character between a pair of double quotes (" ")
    - **For Example:** "Hello World" , "SYBCA division 3 and 4"
6. **The Null Literal**
    - The null type has one value, the null reference, represented by the literal null, which is formed from ASCII characters. A null literal is always of the null type.

NullLiteral:

- **For Example:** null

2.5 Java Keywords

Keywords are reserved words which are used by java programming language for its own feature and functionality. These keywords cannot be used as an identifier.
Java support rich set of default keywords as follow:

abstract	**for**	**new**	**switch**	
assert	**default**	**if**	**package**	**synchronized**
boolean	**goto**	**private**	**this**	
break	**double**	**implements**	**protected**	**throw**
byte	**else**	**import**	**public**	**throws**
case	**enum**	**instanceof**	**return**	**transient**
catch	**extends**	**int**	**short**	**try**
char	**final**	**interface**	**static**	**void**
class	**finally**	**long**	**strictfp**	**volatile**
const	**float**	**native**	**super**	**while**

The keywords const and goto are reserved, even though they are not currently used. This may allow a Java compiler to produce better error messages if these C++ keywords incorrectly appear in programs.

2.6 Java Operators

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

1. Arithmetic Operators
2. Relational Operators
3. Bitwise Operators
4. Logical Operators
5. Assignment Operators
6. Miscellaneous Operators

1. The Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators −

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator.	A + B
- (Subtraction)	Subtracts right-hand operand from left-hand operand.	A – B
*** (Multiplication)**	Multiplies values on either side of the operator.	A * B
/ (Division)	Divides left-hand operand by right-hand operand.	B / A
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	B % A
++ (Increment)	Increases the value of operand by 1.	B++
-- (Decrement)	Decreases the value of operand by 1.	B--

2. The Relational Operators

There are following relational operators supported by Java language.

Operator	Description	Example
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B)
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B)
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B)
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B)
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B)
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B)

3. The Bitwise Operators

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.
Bitwise operator works on bits and performs bit-by-bit operation.

Operator	Description	Example
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B)
| (bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A | B)
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B)
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2

Example:

```
public class UnaryBitWiseOpr{
        public static void main(String[] args){
        int number=-10;
        System.out.print("The decimal number is: ");
        System.out.println(number);
        System.out.print("The Integer to binary of  number is: ");
        System.out.println(Integer.toBinaryString(number));
        int invertednumber=~number;
        System.out.print("The Inverted number is: ");
        System.out.println(invertednumber);
        System.out.print("The Binary of Inverted number is: ");
        System.out.println(Integer.toBinaryString(invertednumber));
        System.out.println("Bit wise shift operator ");
        System.out.println("--------------------------");
        int number1 = -8, number2 = 2, result;
        //8:1000>>2 bit 10
       // bitwise OR between 12 and 25
       result = number1 >> number2;
        //right shift zero fill bitwise operator >>>

        System.out.print("Right shift right shift  bitwise operator >>  ");
       System.out.println(result);//number1>>number2
       System.out.print("Right shift zero fill bitwise operator >>> ");
       System.out.println(number1>>>number2);  // prints 29
       result = number1 << number2;
        System.out.print("Left shift  bitwise operator >>  ");
        System.out.println(result);//number1>>number2
        }
}
```

Output

```
The decimal number is: -10
The Integer to binary of  number is: 11111111111111111111111111110110
The Inverted number is: 9
The Binary of Inverted number is: 1001
Bit wise shift operator
------------------------------------
Right shift right shift  bitwise operator >> -2
Right shift zero fill bitwise operator >>>  1073741822
 Left shift  bitwise operator >> -32
```

4. The Logical Operators

The following table lists the 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

5. The Assignment Operators

Following are the assignment operators supported by Java language:

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator. It adds right operand to the left operand and assigns the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assigns the result to left operand.	C -= A is equivalent to C = C – A
***=**	Multiply AND assignment operator. It multiplies right operand with the left operand and assigns the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides left operand with the right operand and assigns the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Bitwise AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Right shift AND assignment operator.	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

6. Miscellaneous Operators

The Conditional Operator (?:)

The conditional operator is the only ternary operator (three operands) It is used like this:

x > 0 ? x : -x // The absolute value of x

The operands of the conditional operator may be of any type. The first operand is evaluatedand interpreted as a boolean. If the value of the first operand is truthy, then the second operand is evaluated, and its value is returned. Otherwise, if the first operand is falsy, then the third operand is evaluated and its value is returned. Only one of the second and third operands is evaluated, never both.

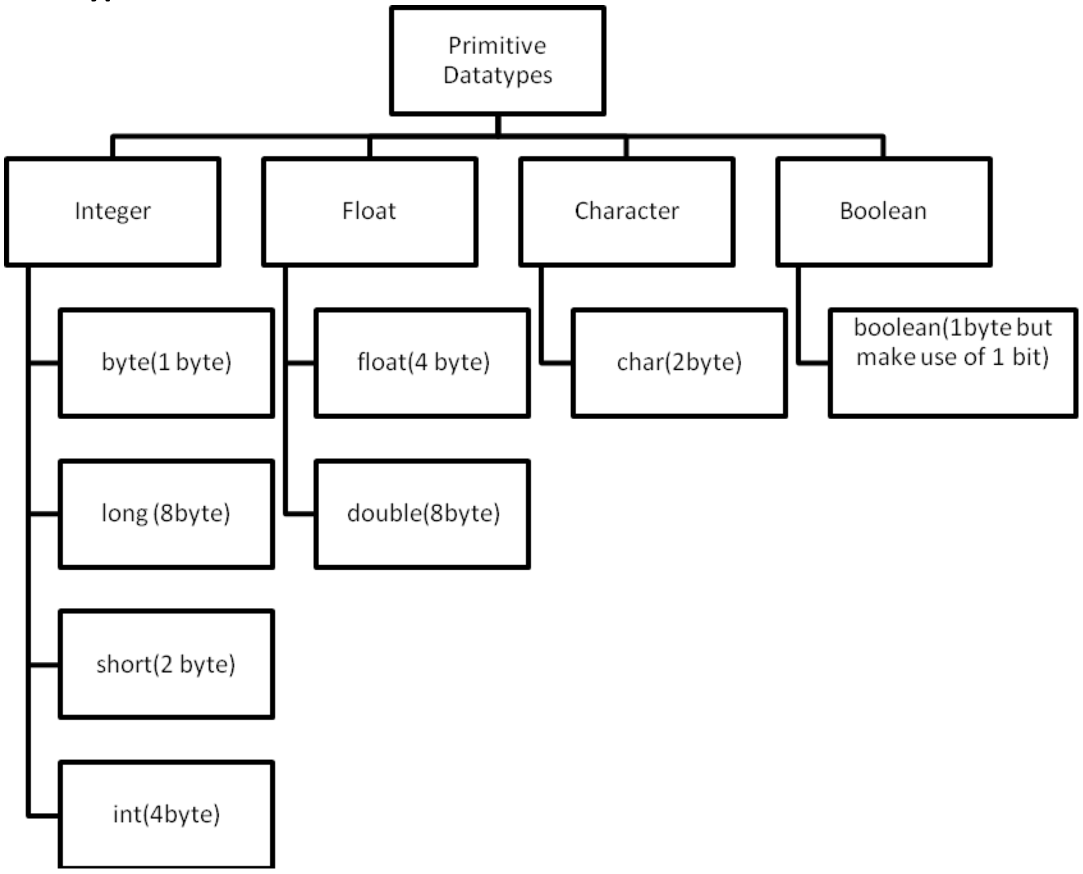
2.7 JAVA Data types

There are majorly two types of languages. First one is **Statically typed language**where each variable and expression type is already known at compile time. Once a variable is declared to be of a certain data type, it cannot hold values of other data types. Example: C,C++, Java. Other, **Dynamically typed languages:** These languages can receive different data types over thetime. Ruby, Python

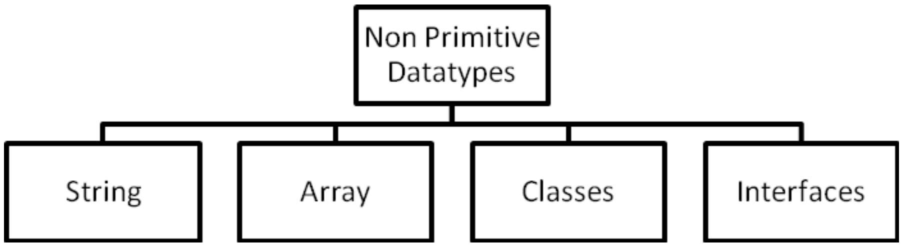
Java is **statically typed and also a strongly typed language** because in Java, each type of data (such as integer, character, hexadecimal, packed decimal, and so forth) is predefined as part of the programming language and all constants or variables defined for a given program must be described with one of the data types.

Following structure show the categories of datatypes in Java

Primitive Datatypes



Nonprimitive Datatypes



Basic Data types Table :

Type	Contains	Default	Size	Range
boolean	True or false	False	1 bit	NA
Char	Unicode Character	\u0000	16 bits	\u0000 \uFFFF
JAVA defines following integer type				
byte	Signed integer 0	0	8 bits	-128 to 127
short	Signed integer 0	0	16 bits	-32768 to 32767
int	Signed integer 0	0	32 bits	-2147483648 to 2147483647
long	Signed integer 0	0	64 bits	-9223372036854775808 to 9223372036854775807
JAVA defines following floating point types				
Float	IEEE 754 floating point 0.0	0.0	32 bits	±1.4E-45 to ±3.4028235E+38
double	IEEE 754 floating point 0.0	0.0	64 bits	±4.9E-324 to ±1.7976931348623157E+308

1. The Boolean Type

boolean data type represents only one bit of information either true or false . Values of type boolean are not converted implicitly or explicitly (with casts) to any other type. But the programmer can easily write conversion code.

```
// A Java program to demonstrate boolean data type
class BooleanType
{
   public static void main(String args[])
   {
      boolean b = true;
      if (b == true)
         System.out.println("BooleanType");
   }
}
```

Output:

BooleanType

2. The char Type

The char data type is a single 16-bit Unicode character. A char is a single character.

- Value: ‘\u0000’ (or 0) to ‘\uffff’ 65535

```
class charDemo
{
      public static void main(String args[])
   {
            char ch1,ch2;
            ch1=88; // ASCII code of X;
            ch2='Y';
            System.out.print("ch1 and ch2 is :");
            System.out.print(ch1+"  " +ch2);
   }
}
```

Note: even though char are not integer in many cases you can operate them as they were integer. class charDemo

```
{
      public static void main(String args[])
   {
            char ch1;
            ch1='X';
            System.out.println("ch1is :"+ch1);
            ch1++;
            System.out.println("How ch1 become now " +ch1);
   }
}
```

JAVA integer type

3. The byte Type

The byte data type is an 8-bit signed two’s complement integer.The byte data type is useful for saving memory in large arrays.

- Size: 8-bit
- Value: -128 to 127

4. The short Type

The short data type is a 16-bit signed two’s complement integer. Similar to byte, use a short to save memory in large arrays, in situations where the memory savings actually matters.

- **Size:** 16 bit
- **Value:** -32,768 to 32,767 (inclusive)

5. **The int Type**

It is a 32-bit signed two's complement integer.

- **Size:** 32 bit
- **Value:** -2^{31} to 2^{31}-1

Note: In Java SE 8 and later, we can use the int data type to represent an unsigned 32-bit integer, which has value in range [0, 2^{32}-1]. Use the Integer class to use int data type as an unsigned integer.

6. **The long Type**

The long data type is a 64-bit two's complement integer.

- Size: 64 bit
- Value: -2^{63} to 2^{63}-1.

Note: In Java SE 8 and later, you can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of 2^{64}-1. The Long class also contains methods like compareUnsigned, divideUnsigned etc to support arithmetic operations for unsigned long.

JAVA float type

7. **The float Type**

The float data type is a single-precision 32-bit IEEE 754 floating point. Use a float (instead of double) if you need to save memory in large arrays of floating point numbers.

- **Size:** 32 bits
- **Suffix :** F/f Example: 9.8f

8. **The double Type**
   a. The double data type is a double-precision 64-bit IEEE 754 floating point. For decimal values, this data type is generally the default choice.
   b. Note: Both float and double data types were designed especially for scientific calculations, where approximation errors are acceptable.
   c. If accuracy is the most prior concern then, it is recommended not to use these data types and use BigDecimal class instead. Please see this for details: Rounding offerrors in Java

```
class EXInteger_Float_Type_Demo
{
   public static void main(String args[])
   {
      // Integer data type is generally used for numeric values
      int i=89;

      // use byte and short if memory is a constraint
      byte b = 4;
      short s = 56;
```

```
        // by default fraction value is double in java
        double d = 4.355453532;

        // for float use 'f' as suffix
        float f = 4.7333434f;

        System.out.println("integer: " + i);
        System.out.println("byte: " + b);
        System.out.println("short: " + s);
        System.out.println("float: " + f);
        System.out.println("double: " + d);
    }
}
```

2.8 Java Variables

Java is a **strongly typed** programming language. This means two things:

1) Every variable in a program must be declared, with a type.
2) Values assigned to a variable should be:
    - same type as the variable's type, or
    - **compatible** with the variable's type

- variable is a container which holds the value while the java program is executed.
- Variable is name of reserved area allocated in memory. In other words, it is a name of memory location. It is a combination of "vary + able" that means its value can be changed.
- A variable provide named storage that programs can use. Each variable has a specific type, which determines the size and layout of the variable's memory

Variable declaration in Java

- You must declare all variables before they can be used.

  data type variable [= value][, variable [= value] ...] ;

- Here data type is one of Java's datatypes and variable is the name of the variable. To declare more than one variable of the specified type, you can use a comma-separated list.

Variable assignment in Java

- From a literal value, to a variable, having compatible types. The statement **a = 100** is one such example.
- From a variable to another variable, of compatible types. The statement **a = c** illustrates this case.
- From a variable expression to a variable. This expression can be formed with variables and literals of compatible types, and is evaluated before the assignment. The statement **c = a + b** below is an example of this.

Example

```
class VariableDemo
{
        public static void main(String args[])
        {
        int a,c;        // Declares three ints, a, b, and c.
        int b = 10;                         // Example of initialization
        a = 10;
        byte B = 22;        // initializes a byte type variable B.
        double pi = 3.14159; // declares and assigns a value of PI.
        char ch = 'a';       // the char variable a iis initialized with value 'a'
         c=a+b;
        System.out.println("Integer type variable a is : "+ a);
        System.out.println("Integer type variable b is : "+ b);
        System.out.println("Byte type variable B is : "+ B);
        System.out.println("Double type variable pi is : "+ pi);
        System.out.println("Char type variable a is : "+ ch);
        System.out.println("Value of variable c is : "+ c);
        }
}
```

2.9 Dynamic initialization

Thought we use constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

```
// Demonstrate dynamic initialization.
class Test
{
         public static void main(String args[])
         {
        double a = 3.0, b = 4.0;
        // c is dynamically initialized
        double c = Math.sqrt(a * a + b * b);
        System.out.println("Hypotenuse is " + c);
         }
}
```

Here, three local variables—a, b,and c—are declared. The first two, a and b, are initialized by constants. However, c is initialized dynamically to the length of the hypotenuse (using the Pythagorean theorem). The program uses another of Java's builtin methods, sqrt( ), which is a member of the Math class, to compute the square root of its argument. The key point here is that the initialization expression may use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

Variable Name Rules

Here are the Java rules for variable names,

A variable name can be a sequence, in any order, of

1) Letters [A-Z, a-z]
2) Numerical digits [0-9]
3) The special characters '$' ("dollar") and '_' ("underscore")

With the following exceptions:

- The name cannot start with a numerical digit [0-9]
- The name must not match with a predefined Java **keyword**

2.10 Types of variables in Java

1. Local variables
2. Instance variables
3. Class/Static variables
4. Constant variable

1. Local Variables

- Local variables are declared in methods, constructors, or blocks. Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block. Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor, or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

Example

Here, age is a local variable. This is defined inside pupAge() method and its scope is limited to only this method.

```
public class Test {
  public void Age() {
    int a = 0;
    a = a + 7;
    System.out.println("Age is : " + a);
  }
  public static void main(String args[]) {
    Test test = new Test();
    test.Age();
  }
}
```

This will produce the following result –

Output

```
Age is: 7
```

2. Instance Variables

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

- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However, visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers, the default value is 0, for Booleans it is false, and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods (when instance variables are given accessibility), they should be called using the fully qualified name. **ObjectReference.VariableName.**

For Example

```
import      java.io.*;
public class Faculty {
  // this instance variable is visible for any child class.
  public String name;
  // salary  variable is visible in Employee class only.
  private double salary;
  // The name variable is assigned in the constructor.
  public Faculty (String empName) {
    name = empName;
  }
  // The salary variable is assigned a value.
  public void setSalary(double fSal) {
    salary = fSal;
  }
  // This method prints the employee details.
  public void printFdata() {
    System.out.println("name  : " + name );
    System.out.println("salary :" + salary);
  }
  public static void main(String args[]) {
    Faculty Ob = new Faculty ("Krupali Patel");
    Ob.setSalary(50000);
    Ob.printFdata();
  }
}
```

This will produce the following result −

Output

```
name : Krupali Patel
salary :50000.0
```

3. **Class/Static Variables**

- Static variables are also known as class variables.
  Static or class variables are declared with the static keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final, and static. Constant variables never change from their initial value.
- Static variables are stored in the static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor.
- Additionally, values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name **ClassName.VariableName**.

For Example

```
import      java.io.*;
public class Faculty {
  // salary variable is a private static variable
  private static double salary;
  // DEPARTMENT is a constant
  public static final String DEPARTMENT = "BCA ";
  public static void main(String args[]) {
    salary = 50000;
    System.out.println(DEPARTMENT + "average salary:" + salary);
  }
}
```

This will produce the following result −

Output

```
BCA average salary:1000
```

4. **Static Variables**
    - As a name suggest constant is a variable whose value can not changed.
    - When declaring class variables as public static final, then variable names (constants) are all in upper case. If the static variables are not public and final, the naming syntax is the same as instance and local variables.

The syntax to declare a constant is as follows:

static final datatype identifier_name=value;

For Example:

static final double WEIGHT=42.78;

Where ,static and final are the non-access modifiers. The double is the data type and WEIGHT is the identifier name in which the value 42.78 is assigned.

- In the above statement, the static modifier causes the variable to be available without an instance of its defining class being loaded and the final modifier makes the variable fixed.
- Here a question arises that why we use both static and final modifiers to declare a constant?
- If we declare a variable as static, all the objects of the class (in which constant is defined) will be able to access the variable and can be changed its value. To overcome this problem, we use the final modifier with a static modifier.
- When the variable defined as final, the multiple instances of the same constant value will be created for every different object which is not desirable.
- When we use static and final modifiers together, the variable remains static and can be initialized once. Therefore, to declare a variable as constant, we use both static and final modifiers. It shares a common memory location for all objects of its containing class.

2.11 Scope or lifetime of variable

Java allows variables to be declared within any block. A block is begun with an opening curly brace and ended by a closing curly brace. A block defines a scope. Thus, each time you start a new block, you are creating a new scope. As you probably know from your previous programming experience, a scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects. Variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope. Therefore, variables declared within a method will not hold their values between calls to that method. Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope. If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered. For example, consider this program:

For Example

```
class LifeTime {
 public static void main(String args[]) {
 int x;
 for(x = 0; x < 3; x++) {
```

```
 int y = -1; // y is initialized each time block is entered
 System.out.println("y is: " + y); // this always prints -1
 y = 100;
 System.out.println("y is now: " + y);
 }
 }
}
```

The output generated by this program is shown here:

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

As you can see, y is always reinitialized to –1 each time the inner for loop is entered. Even though it is subsequently assigned the value 100, this value is lost. One last point: Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope. In this regard, Java differs from C and C++. Here is an example that tries to declare two separate variables with the same name. In Java, this is illegal. In C/C++, it would be legal and thetwo bars would be separate.
For Example

```
// This program will not compile
class ScopeErr
{
 public static void main(String args[]) {
 int bar = 1;
 { // creates a new sope
 int bar = 2; // Compile-time error – bar already defined!
}
}
```

2.12 Java Array

- Java array is an object which contains elements of a similar data type.
- An array represents the group of element of same data type and also it reduces number of statements into 1-3 statements.
- Array is a continues block of memory to store same type of values to store the same type of value we create a array object.
- To create an array object we need to create array reference variable. Array reference variable used to store the address of the object.
- It is a fixed length cannot modify again. We can access array unit with the help of index value.

- Index value always starts with zero and ends with array reference variable-1 i.e., e., array reference **variable.length.**

Obtaining an array

- Obtaining an array is a two-step process.
    1. You must declare a variable of the desired array type.

You must allocate the memory that will hold the array, using new, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated.

syntax:

```
<datatype><name of the array>[]= new <datatype>[<size of the array>];
```

example:

```
int arr[]= new int [100];//creation of array
```

Types of Array

1. One dimensional array
2. Multidimensional array

One Dimensional Array

- A one-dimensional array is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type.
- The general form of a one dimensional array declaration is
  type var-name[ ];
- Here, type declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines what type of data the array will hold.
- For example, the following declares an array named game_players with the type "array of int":
  int game_players[];
- Although this declaration establishes the fact that game_players is an array variable, no array actually exists. In fact, the value of game_players is set to null, which represents an array with no value. To link game_players with an actual, physical array of integers, you must allocate one using new and assign it to game_players. new is a special operator that allocates memory.
- The general form of new as it applies to one-dimensional arrays appears as follows:
  array-var = new type[size];
- Here, type specifies the type of data being allocated, size specifies the number ofelements in the array, and array-var is the array variable that is linked to the array. That is, to use new to allocate an array, you must specify the type and number of elements toallocate. The elements in the array allocated by new will automatically be initialized to zero.
- This example allocates a 6-element array of integers and links them to game_players.
  game_players = new int[6];
- After this statement executes, game_players will refer to an array of 6 integers. Further, all elements in the array will be initialized to zero. Obtaining an array is a two-step process. Thus, in Java all arrays are dynamically allocated. Once you have allocated an

array, you can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero.

- For example,we have different games in order like Chess, Cricket, Kabaddi, Hockey, Vollyball and Khokho this statement assigns the value 11 to the second element of game_players.

```
game_players[1] = 11;
```

The next line displays the value stored at index 3.

```
System.out.println(game_players[3]);
```

Putting together all the pieces, here is a program that creates an array of the number of players in each game.

For Example: a one-dimensional array.

```
class Array {
 public static void main(String args[]) {
 int game_players[];
 game_players = new int[6];
 game_players[0] = 1;
 game_players[1] = 11;
 game_players[2] = 7;
 game_players[3] = 11;
 game_players[4] = 6;
 game_players[5] = 9;
  System.out.println("Hockey has " + game_players[3] + " players.");
 }
}
```

- When you run this program, it prints the number of players in Hockey. As mentioned, Java array indexes start with zero, so the number of players in Hockey is game_players[3] or 11.
- It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here:

int game_players[] = new int[6];

- This is the way that you will normally see it done in professionally written Java programs.
- Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types.
- An array initializer is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements.
- The array will automatically be created large enough to hold the number of elements you specify in the array initializer.
- There is no need to use **new.** For example, to store the number of players in each game, the following code creates an initialized array of integers:

```
class AutoArray
{
    public static void main(String args[]) {
     int game_players[] = { 1, 11, 7, 11, 6, 9};
    System.out.println("Hockey has " + game_players[3] + " players.");
     }
}
```

- When you run this program, you see the same output as that generated by the previous version. Java strictly checks to make sure you do not accidentally try to store or reference values outside of the range of the array. The Java run-time system will check to be sure that all array indexes are in the correct range. (In this regard, Java is fundamentally different from C/C++, which provide no run-time boundary checks.) For example, the run-time system will check the value of each index into game_players to make sure that it is between 0 and 5 inclusive. If you try to access elements outside the range of the array (negative numbers or numbers greater than the length of the array), you will cause a runtime error.
- It finds the average of a set of numbers.// Average an array of values.

```
class Average {
 public static void main(String args[]) {
 double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
 double result = 0;
 int i;
 for(i=0; i<5; i++)
 result = result + nums[i];
 System.out.println("Average is " + result / 5);
 }
}
```

Multi Dimensional Array

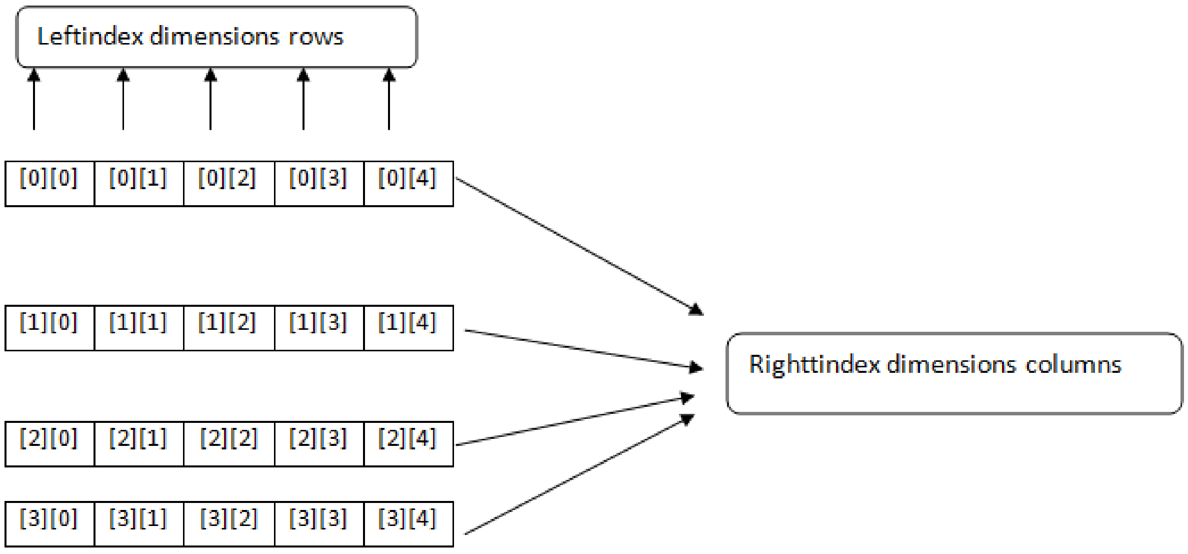
A multidimensional array is an array with more than two dimensions called 2D for two dimention and 3D for three dimensional ans so on.

In a matrix, the two dimensions are represented by rows and columns. Each element is defined by two subscripts, the row index and the column index.

For example, the following declares a two-dimensional array variable called twoD.

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

This allocates a 4 by 5 array and assigns it to twoD. Internally this matrix is implemented as an array of arrays of int. Conceptually, this array will look like the one shown in Figure



(figure: structure of two dimensional array)

The following program numbers each element in the array from left to right, top to bottom, and then displays these values:

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

This program generates the following output:

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

Alternative Array Declaration Syntax

There is a second form that may be used to declare an array:

type[ ] var-name;

Here, the square brackets follow the type specifier, and not the name of the array variable. For example, the following two declarations are equivalent:

int al[] = new int[3];

int[] a2 = new int[3];

The following declarations are also equivalent:

char twod1[][] = new char[3][4];

char[][] twod2 = new char[3][4];

This alternative declaration form is included mostly as a convenience.

2.13 Type Conversion and Casting

it is quite common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an int value to a long variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no conversion defined from double to byte. Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a cast, which performs an explicit conversion between incompatible types. Let's look at both automatic type conversions and casting.

Java's Automatic Conversions

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

1. The two types are compatible.
2. The destination type is larger than the source type.

When these two conditions are met, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement isrequired. For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, the numeric types are not compatible with char or boolean. Also, char and boolean are not compatible with each other. As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, or long.

Casting Incompatible Types

Although the automatic type conversions are helpful, they will not fulfill all needs. For example, what if you want to assign an int value to a byte variable? This conversion will not be performed automatically, because a byte is smaller than an int. This kind of conversion is sometimes called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into

the target type. To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion.
It has this general form:
(target-type) value
Here, target-type specifies the desired type to convert the specified value to. For example, the following fragment casts an int to a byte. If the integer's value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range.

```
int a; byte b;
// ...
b = (byte) a;
```

A different type of conversion will occur when a floating-point value is assigned to an integer type: truncation. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost. For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23will have been truncated. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

The following program demonstrates some type conversions that require casts:
For Example

```
 class Type_Conversion {
 public static void main(String args[]) {
 byte b;
 int i = 257;
 double d = 323.142;
 System.out.println("\\nConversion of int to byte.");
 b = (byte) i;
 System.out.println("i and b " + i + " " + b);
 System.out.println("\\nConversion of double to int.");
 i = (int) d;
 System.out.println("d and i " + d + " " + i);
 System.out.println("\\nConversion of double to byte.");
 b = (byte) d;
 System.out.println("d and b " + d + " " + b);
 }
}
```

This program generates the following output:

```
Conversion of int to byte.
i and b 257 1
Conversion of double to int.
d and i 323.142 323
Conversion of double to byte.
d and b 323.142 67
```

Let's look at each conversion. When the value 257 is cast into a byte variable, the result is the remainder of the division of 257 by 256 (the range of a byte), which is 1 in this case. When the d is converted to an int, its fractional component is lost. When d is converted to a byte, its fractional component is lost, and the value is reduced modulo 256, which in this case is 67.

Automatic Type Promotion in Expressions
In addition to assignments, there is another place where certain type conversions may occur: in expressions. To see why, consider the following. In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand.
For example, examine the following expression:
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
The result of the intermediate term a * b easily exceeds the range of either of its byte operands. To handle this kind of problem, Java automatically promotes each byte or short operand to int when evaluating an expression. This means that the subexpression a * b is performed using integers—not bytes. Thus, 2,000, the result of the intermediate expression, 50
* 40, is legal even though a and b are both specified as type byte. As useful as the automatic promotions are, they can cause confusing compile-time errors.

For example, this seemingly correct code causes a problem:
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!

The code is attempting to store 50 * 2, a perfectly valid byte value, back into a byte variable. However, because the operands were automatically promoted to int when the expression was evaluated, the result has also been promoted to int. Thus, the result of the expression is now of type int, which cannot be assigned to a byte without the use of a cast. This is true even if, as in this particular case, the value being assigned would still fit in the target type.

In cases where you understand the consequences of overflow, you should use an explicit cast, such as
byte b = 50;
b = (byte)(b * 2);

which yields the correct value of 100.

The Type Promotion Rules
In addition to the elevation of bytes and shorts to int, Java defines several type promotion rules that apply to expressions. They are as follows. First, all byte and short values are promoted to int, as just described. Then, if one operand is a long, the whole expression is promoted to long.If one operand is a float operand, the entire expression is promoted to float. If any of the

operands is double, the result is double. The following program demonstrates how each value in the expression gets promoted to match the second argument to each binary operator:

```
class Promote {
 public static void main(String args[]) {
 byte b = 42;
 char c = 'a';
 short s = 1024;
 int i = 50000;
 float f = 5.67f;
 double d = .1234;
 double result = (f * b) + (i / c) - (d * s);
System.out.println((f * b) + " + " + (i / c) + " - " + (d *
s));
 System.out.println("result = " + result);
 }
}
```

Let's look closely at the type promotions that occur in this line from the program:
double result = (f * b) + (i / c) - (d * s);
In the first subexpression, f * b, b is promoted to a float and the result of the subexpression is float. Next, in the subexpression i / c, c is promoted to int, and the result is of type int. Then, in d * s, the value of s is promoted to double, and the type of the subexpression is double. Finally, these three intermediate values, float, int, and double, are considered. The outcome of float plus an int is a float. Then the resultant float minus the last double is promoted to double,which is the type for the final result of the expression.

2.14 Garbage Collection

- Java has automatic garbage collection as e objects are dynamically allocated by using the new operator, let’s see how objects are destroyed and their memory released for later reallocation.
- In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator.
- Java takes a different approach; it handles deallocation for you automatically.
- The technique that accomplishes this is called **garbage collection**.
- It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- There is no explicit need to destroy objects like C++.
- Garbage collection only occurs periodically during the execution of your program. It will not occur `simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

2.15 The finalize( ) Method

- Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed. To handle
- such situations, Java provides a mechanism called finalization. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
- To add a finalizer to a class, you simply define the finalize( ) method. The Java run time calls that method whenever it is about to recycle an object of that class.
- Inside the finalize( ) method you will specify those actions that must be performed before an object is destroyed.
- The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects.
- Right before an asset is freed, the Java run time calls the finalize( ) method on the object.
- The finalize( ) method has this general form:

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

- Here, the keyword protected is a specifier that prevents access to finalize( ) by code defined outside its class.
- It is important to understand that finalize( ) is only called just prior to garbage collection. it is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—finalize( ) will be executed. Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on finalize( ) for normal program operation.

2.16 Command-Line Arguments

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

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

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

```
import java.io.*;
import java.lang.*;
class cmdLine
{
	public static void main(String args[])
	{
		for(int i=0;i<args.length;i++)
		{
		System.out.println("args["+i+"] is: "+args[i]);}
	}
}
```

Try executing this program, as shown here:

```
C:\java>java cmdLine This is Command line Example 1
args[0] is: This
args[1] is: is
args[2] is: Command
args[3] is: line
args[4] is: Example
args[5] is: 1
```

Note: pass arguments with comma separated values which of type String. if you want to take numeric values then perform type conversion. For Example following program take numbers as command line arguments and perform arithmetic operations

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

	}
//before type conversion
	System.out.println("Numeric argunments by defalut type of String ,Concatination is : 
"+args[0]+args[1]);
//after type conversion
	 int x=Integer.parseInt(args[0]);
	 int y=Integer.parseInt(args[1]);
	 System.out.println("Numeric argunments converted into int then perform addion is : 
"+(x+y));
		}
}
```

Output:

```
D:\java>java cmdLine  3 4
args[0] is: 3
args[1] is: 4
Numeric argunments by defalut type of String ,Concatination is : 34
Numeric argunments converted into int then perform addion is : 7
```

<u>2.17</u> <u>Control Structure</u>

To control the flow of execution of a program programming language uses control statements. In Java programming Language, we can control the flow of execution of a program based on some conditions. Java control statements can be put into the following three categories: selection, iteration, and jump.

Conditional Control Statements/Selection Statements: These statements allow you to control the flow of your program's execution based upon conditions known only during run time. The selection statements allow your program to choose a different path of execution based on a certain condition.

Looping Control/Iterative Statement: There may be a situation when you need to execute block of code more than one times. Iteration (Looping) statements allow program execution to repeat one or more statements.

Sequential Control/Jumping Statements: Jump statements transfer the control to other parts of the program.

Following is General form of Java Control structure

Selection Statement /Conditional Control Statements		
Statement	**Description and Syntax**	**Example**

The if Statement:	If the Boolean expression evaluates to true then the block of code inside the if statement will be executed. If not the first set of code after the end of the if statement (after the closing curly brace) will be executed.
	Syntax :
	if(Condition)
	{
	//Statements will execute if the Condition is true
	}
		public class Test {
		public static void main(String args[]){
		int x = 10;
		if(x < 20){
		System.out.print("This is if statement");
		}
		}
		}

The if...else Statement	An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.
	Syntax :
	if(Condition){
	//Executes when the Condition is true
	}else{
	//Executes when the Condition is false
	}
		public class Test {
		public static void main(String args[]){
		int a = 40;
		
		if(a < 10){
		System.out.print("This is if statement");
		}else{
		System.out.print("This is else statement");
		}

		} }

The if...else if...else Statement	An if statement can be followed by an optional *else if...else* statement, which is very useful to test various conditions using single if...else if statement.
	Syntax :
	if(Condition 1)
	{
	//Executes when the Condition 1 is true
	}
	else if(Condition 2)
	{
	//Executes when the Condition 2 is true
	}
	else if(Condition 3)
	{
	//Executes when the Condition 3 is true
	}
	else
	{
	//Executes when the none of the above condition is true.
	}
		public class Test {
		public static void main(String args[]){
		int a = 50;
		if(a == 10)
		{
		System.out.print("Value of a is 10");
		}
		else if(a == 20)
		{
		System.out.print("Value of a is 20");
		}
		else if(a == 50)
		{
		System.out.print("Value of a is 50");
		}
		else
		{
		System.out.print("This is else statement");
		}
		}
		}

Nested if...else Statement	**Syntax :**
	if(Condition 1)
	{
	//Executes when the Condition 1 is true
	if(Condition 2)
	{
	//Executes when the Condition 2 is true
	}
	}
		public class Test {
		public static void main(String args[]){
		int a = 2;
		int b = 5;
		if(a == 2)
		{
		if(b == 5)
		{
		System.out.print("a = 2 and b = 5");
		}
		}
		}
		}

The switch Statement	A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked	public class Switch_Case {
		public static void main(String args[]){
		char grade = args[0].charAt(0);
		//char grade = 'C';

	for each case.
	Syntax :
	switch(expression)
	 {
	   case case1 :
	         //Statements
	     break; //optional
	   case case2 :
	     //Statements
	     break; //optional
	    //You can have any number of case statements.
	   default : //Optional
	     //Statements
	}
		switch(grade)
		{
		  case 'A' :
		    System.out.println("Distinction!");
		    break;
		  case 'B' :
		  case 'C' :
		    System.out.println("First class ");
		    break;
		  case 'D' :
		    System.out.println("You passed");
		  case 'F' :
		    System.out.println("Fail");
		    break;
		  default :
		    System.out.println("Invalid grade");
		}
		System.out.println("Your grade is " + grade);
		  }
		}

Looping Control/Iterative Statement:

Statement	Description and Syntax	Example

while loop	A while loop statement in Java programming language repeatedly executes a target statement as long as a given condition is true.
	Syntax :
	while(Boolean_expression)
	{
	  // Statements
	}
		public class Test {
		  public static void main(String args[]) {
		    int a = 1;
		    while(a < 10) {
		    System.out.print("value of a : " + a);
		    a++;
		    System.out.print("\n");
		    }
		  }
		}

for loop	A for loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times.
	Syntax :
	for(initialization; Boolean_expression; increment) {
	  // Statements
	}
		public class Test {
		  public static void main(String args[]) {
		    for(int a = 10; a <=20; a = a + 1)
		{//a++
		      System.out.print("value of a : " + a
		);
		      System.out.print("\n");
		    }
		  }

		}

do...while loop	A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.
	Syntax :
	do
	{
	// Statements
	}while(Boolean_expression);
		public class Test {
		public static void main(String args[]) {
		int a = 10;
		do {
		System.out.print("value of a : " + a);
		a++;
		System.out.print("\n");
		}while(a < 20);
		}
		}

Loop Control Statement		

Break	break statement used to break the sequential execution of loop
	Syntax :
	break;
		public class Test {
		public static void main(String args[]) {
		int [] numbers = {1,2,3,4,5};
		for(int a : numbers) {
		if(a == 3) {
		break;
		}
		System.out.print(a);
		System.out.print("\n");
		} }}

continue	The continue keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.
	• In a for loop, the continue keyword causes control to immediately jump to the update statement.
	• In a while loop or do/while loop, control immediately jumps to the Boolean expression.
		public class Test {
		public static void main(String args[]) {
		int [] numbers = {1,2,3,4,5};
		for(int a : numbers) {
		if(a == 3) {
		continue;
		}
		System.out.print(a);
		System.out.print("\n");
		}
		}
		}

Jumping Statements / Sequential Control		
Statement	**Description and Syntax**	**Example**

continue	The continue keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.
	• In a for loop, the continue keyword
		public class Test {
		public static void main(String[] args)
		{

	causes control to immediately jump to the update statement.
	• In a while loop or do/while loop, control immediately jumps to the Boolean expression.
		`// label for outer loop`
		`outer:`
		`for (int i = 0; i < 10; i++) {`
		`for (int j = 0; j < 10; j++)`
		`{`
		`if (j == 1)`
		`break outer;`
		`System.out.println("`
		`value of j = " + j);`
		`}`
		`} // end of outer loop`
		`} // end of main()`
		`}`

Label	The only place where a label is useful in Java is right before nested loop statements. We can specify label name with break to break out a specific outer loop. Labels are where you can shift control from break and continue.	
Note: Java does not support goto, it is reserved as a keyword just in case they wanted to add it to a later version.		

<u>2.18</u> <u>Exercise</u>

1. Is the following statement true or false? If it is false then supply suitable example:“The data is lost when converted from higher data type to lower data type.”
2. Can you make the keyword public to private in main() method? If yes, what will happen?
3. Explain with an example data types boolean and byte in Java.
4. Explain with an example data type byte in Java.
5. What is type conversion in Java? Explain with an example.
6. Write the difference between the operators = and = = in Java.
7. What is type casting? Give an example to explain it.
8. What are literals? State the boolean literals.
9. List the different operators in Java.
10. What are literals in Java? Mention their different types.
11. What are different integer data types in Java?
12. Explain any two logical operators in Java with example.
13. What is meant by an assignment statement? For what purpose it is used?
14. State the bitwise operators in Java. Explain any one.
15. Explain how arrays are created in Java? How array elements are accessed?
16. How do you declare an array in Java? Give an example.
17. How do you declare a two dimensional array in Java? Give an example.
18. Define an array. How do you declare and create array objects?
19. What is the difference between static and non-static variables?
20. What are destructors?
21. How is object destruction done in Java?
22. What is Garbage collection?
23. What is the use of finalize method?
24. Compare Garbage collection and finalize method?
25. How is it guaranteed that finalize methods are called?

26. What is CommandLine Arguments.
27. Differentiate break and label break.
28. Explain switch statement in Java.
29. Explain switch statement with example in Java.
30. Explain the goto statement in Java.
31. Explain while loop in Java.
32. Explain any one loop statement in Java with an example.
33. Explain Conditional Control in java

3 JAVA Classes, Methods & Objects

3.1 Introduction

A class is a template for an object. And an object is an instance of a class. All program activities in JAVA occurs within a class means which you define object.Thus an understanding of a class is critical to successful java programming.

3.2 Simple class structure

Definition: " A class is a user defined data type with a template hat serve to define its properties"

- Class is collection of an instance variables and methods.
- Class is declared by class keyword.
- Class may contain variables, methods and constructors.
- Variables represent its state.
- Methods provide the logic that constitutes the behavior defined by a class.
- There can be both static and instance variables and methods.
- Constructor initializes the state of a new instance of a class.

Syntax:

3.3 Objects

An object is an instance of a class.

Object − Objects have states and behaviors. Example: A table has states - color, name, height ,weight etc.

3.3.1 To create object

As mentioned previously, a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the new keyword is used to create new objects.There are three steps when creating an object from a class −

Declaration − A variable declaration with a variable name with an object type.

Instantiation − The 'new' keyword is used to create the object.

Initialization − The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

3.3.2 Declaring object

Declaration − A variable declaration with a variable name with an object type.

3.3.3 The new operator

When you are declaring a class in java, you are just creating a new data type. A class provides the blueprint for objects. You can create an object from a class. However obtaining objects of a class is a two-step process :

1. **Declaration :** First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object. Below is general syntax of declaration with an example :
   Syntax :
   class-name var-name;
   Example :
   // declare reference to an object of class Box
   Box mybox;

A variable in this state, which currently references no object, can be illustrated as follows (the variable name, mybox, plus a reference pointing to nothing):

2. **Instantiation and Initialization :** Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the new operator. The new operator instantiates a class by dynamically allocating (i.e, allocation at run time) memory for a new object and returning a reference to that memory. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.
   - The new operator is also followed by a call to a class constructor, which initializes the new object. A constructor defines what occurs when an object of a class is created. Constructors are an important part of all classes and have many significant attributes. In below example we will use the default constructor. Below is general syntax of instantiation and initialization with an example :

Syntax:

```
        var-name = new class-name();
```

Example :

```
        // instantiation via new operator and initialization via default constructor of class Box
        mybox = new Box();
```

<u>3.3.4 Assigning object reference variable</u>

We are assign one object reference variable to another object reference variable, we are not creating a copy of the object, we are only making a copy of the reference. For example, in the following program, the object references for class Box are mybox1 and mubox2. We can create multiple references to the same object.

For Example

```
        Box mybox1=new Box();
                Box mybox2=new Box();

                mybox1.width=10;
                mybox1.height=20;
                mybox1.depth=15;

                mybox2.width=3;
                mybox2.height=6;
                mybox2.depth=9;
```

<u>3.3.5 This keyword</u>

1. It is used to refer to the current object.
2. It is used to resolve the any name space collisions that might occur between instance variable and local variables.
3. Used to overcome the instance variable hiding.

3.4 Methods

The method is a large one because JAVA gives them so much power and flexibility.

Syntax:

```
Type method_name(parameter_list)
{
        //body of the method
}
```

- Here, type specifies the type of data return by the method. These can be any valid type including class type that you create.
- If the method does not return a value its return type must be **void**
- The name of the method is specify by the name , these can be any legal identifier other than those already used by other item within the current scope.
- The parameter list is a sequence of type and identifier paired separated by commas.
- Parameters are variable that receive that value of the argument passed to the method when it is called.
- If the method has no parameter then parameter list will be empty.
- Method that have a return type other than void return a value that is return value.

```
class Box
{
 double  width;
 double height;
 double depth;
         void volume()
        {
         System.out.println("Volume is:");
         System.out.println(width*height*depth);
        }
}
class BoxDemo
{
        public static void main(String args[])
        {
         Box mybox1=new Box();
         Box mybox2=new Box();

         mybox1.width=10;
         mybox1.height=20;
         mybox1.depth=15;

         mybox2.width=3;
         mybox2.height=6;
         mybox2.depth=9;
```

```
            mybox1.volume();
        mybox2.volume();

            }

    }
```

```
Output:
E:\java>javac BoxDemo.java
E:\java>java BoxDemo
Volume is:
3000.0
Volume is:
162.0
```

3.4.1 Constructor

- Objects that will require some form of initialization when it is create.
- To accommodate this feature JAVA programming allows you to define constructor for your class.
- A constructor is special methods that create and initialize an object of a particular class.
- It has the same name as it's class name and it may accept argument.
- A constructor does not have a return type instead of a constructor return a reference to an object that is create.
- If you do not explicitly declare a constructor for a class, the JAVA compiler automatically generates a default constructor that has no arguments.
- A constructor is never call directly instead of it is invoked via the new Operator
- Constructors have no return type the implicit return type of a class constructor is the class type itself.
- It is the constructor's job to initialize the internal state of an object.

3.4.2 Parameterized Constructor

A constructor that accept arguments and uses them to initialized it's instance variable is called parameterized constructor. If we need to construct Box object of various dimension we can accommodate these by parameterized constructor.

Write a program that uses a parameterized constructor to initialize the dimension of a Box.

```
class Box
{
        double  width;
        double height;
        double depth;
       //this is a construcot of Box class
       Box(double w,double h,double d)
       {
        width=w;
```

```
                 height=h;
                 depth=d;
                }
                double volume()
                {
                 return width*height*depth;
                }
}
class Box_Param_Cons_Demo
{
                public static void main(String args[])
                {
                 double vol1,vol2;
                 Box mybox1=new Box(10,20,15);
                 Box mybox2=new Box(6,3,9);
                 vol1=mybox1.volume();
                  System.out.println("Box Volume is:"+vol1);
                 vol2=mybox2.volume();
                 System.out.println("Box Volume is:"+vol2);
                }
}

/*Output:
E:\java>javac Box_Param_Cons_Demo.java
E:\java>java Box_Param_Cons_Demo
Box Volume is:3000.0
Box Volume is:162.0
*/
```

3.4.3 Constructor overloading

- A class may have several constructors these feature is called constructor overloading.
- When constructors are overloaded each ha still called the name of it's class.
- It must have a different parameter list. The signature of each constructor must be different.
- The signature is a constructor with combination of its name and the sequence of its parameter type.
- If two constructors in the same class have same signature this represent an ambiguity. A JAVA compiler generates an error message because it is unable to determine which form of constructor to be used.
- Constructor overloading provide several ways to create an object of particular class.
- An object that has multiple instance variables, in some situation you need to assign default values to all of these variables, in other situation required that specific values assign to the remaining variables.

- These can be done by providing multiple constructors for the class. it allows you to create and initialize an object by using different types of data.

Write a program to demonstrate constructor overloading.

```
class Rect
{
 int len;
 int h;
            Rect(int l,int b) //Constructor1
            {
             len=l;
             h=b;
            }
            Rect(int x) //Constructor2
            {
             len=h=x;
            }
int Area()
{
            return (len*h);
}
}
class Cons_overloading_Demo
{
            public static void main(String args[])
            {
             Rect r1=new Rect(5,10);
             int res1=r1.Area();
             System.out.println("Area of Rectangle is :-"+res1);
             Rect r2=new Rect(6);
             int res2=r2.Area();
             System.out.println("Area of Rectangle is :-"+res2);


            }
}
/*
Output:
E:\java>javac Cons_overloading_Demo.java
E:\java>java Cons_overloading_Demo
Area of Rectangle is :-50
Area of Rectangle is :-36
*/
```

3.5 This keyword

Sometimes a method will need to refer to the object that invoked it. To allow these feature JAVA Programming defines the this keyword. this can be use any method to refer to the current object. this is always a reference to the object on which the method was invoke.
You can use this anywhere a reference to an object of the current class type is permitted.
Example:

```
//a redundant use of this keyword.
Box(double w, double h, double d) //Construcor
{
 this.width=w;
 this.height=h;
 this.depth=d;
}
```

The version of Box() operators exactly like the earlier version. The use if this is redundant but perfectly correct,
Inside Box() this will always refer to the invoking object. To hide the local variable to instance variable this keyword is used.

3.6 Instance variable hiding

- When a local variable has the same name as an instance variable then the local variable hide the instance variable.
- Here this keyword overcome the instance variable hiding
- The instance variable hiding variable cohesion.

3.7 Super keyword

- It returns the reference of base class object.
- super can access middle variable overridden method or invoke super class constructor.
- A class inherits the state and behavior define by all it's super class state is determined by variable behavior and determined by methods
- An object has one copy of every instance variable define not only by it's class but also by every super class of its class.
- A static or instance variable in a sub class my have the same name as a super class variable.
- The variable hides the super class variable.
- It is possible to access a hidden variable by using the super keyword.

Syntax:

super.variable_name;

- Variable_name: is name of the variable in the super class.
- The super keyword enables to access the super class variable and construct a sub class by the same variable name
  Example:
  class my_super_class

```
{
        int i=100;
}
class my_sub_class extends my_super_class
{
        int i=200;
        void display()
        {
         System.out.println("Value of i sub class is:-"+i);
         System.out.println("Value of i super class is:-"+super.i);
        }
}
class Super_keyword_Demo
{
        public static void main(String args[])
        {
         my_sub_class obj=new my_sub_class();
         obj.display();
        }
}
```

Output:

```
E:\java>javac Super_keyword_Demo.java

E:\java>java Super_keyword_Demo
Value of i sub class is:-200
Value of i super class is:-100
```

<u>3.8</u> <u>Method overloading</u>

- In JAVA it is possible to define two or more methods within the same class that share the same name and parameter declarations are different. When these is the case the methods are said to be overloaded, and the process id referred to as method overloading.
- Method overloading is one of the ways that JAVA implements polymorphism.
- Method overloading is allow you to use same name for a group of methods that basically have the same purpose.
- The println() is the good example of this concept. It has several overloaded form.
- Each of these accept an argument of different type, the type may be Boolean, chra, Int, long, double, short, char[], string or object.
- It is much more convenient for programmers to remembers one method name rather than several different once.
- Another advantage of method overloading is that it provides an easy different input and output parameters.

- JAVA matches up the method name first and then the number and type of parameter to decide which one of the definitions to execute polymorphism.
  Example:
```
class OverloadMethod
{
        void test()
        {
         System.out.println("No arguments");
        }
        // overload test for one integer parameter
        void test(int a)
        {
         System.out.println("a="+a);
        }
        // overload test for two integer parameter
        void test(int a,int b)
        {
         System.out.println("a="+a+" and b="+b);
        }
        // overload test for one double parameter
        double test(double d)
        {
         System.out.println("double d="+d);
         return d*d;
        }
}
class OverloadDemo
{
        public static void main(String args[])
        {
         double result;
         OverloadMethod obj=new OverloadMethod();
         //call versions of test
         obj.test();
         obj.test(10);
         obj.test(10,20);
         result=obj.test(125.20);
        }
}
```
  Output:
```
E:\java>javac OverloadDemo.java

E:\java>java OverloadDemo
No arguments
```

```
a=10
a=10 and b=20
double d=125.2
```

3.9 Call by reference call by value

There are two ways that a computer language can pass and argument to a subroutine

1. Call by value
2. Call by reference

3.9.1 Call by value:

- This method copy value of an argument into the formal parameter of the subroutine.
- Changes made to the parameter of the subroutine have no effect on the argument
- When you pass a simple type to a method it is pass by value. thus what occur to the parameter that receive the argument that has no effect out side the method.
- In case of call by value original value is not changed. Let's take a simple example:

Example:

```
class Operation{
 int data=50;

 void change(int data){
 data=data+100;//changes will be in the local variable only
 }

 public static void main(String args[]){
   Operation op=new Operation();

   System.out.println("before change "+op.data);
   op.change(500);
   System.out.println("after change "+op.data);

 }
}
```

Output:

```
before change 50
after change 50
```

3.9.2 Call by reference:

- In case of call by reference original value is changed if we made changes in the called method. If we pass object in place of any primitive value, original value will be changed.
- A reference is an argument (not the value of the argument) is passed to the parameter.
- Inside the subroutine these reference is used to access the actual argument specified in the call.

- This means that changes means to parameter will affect the argument used to call the subroutine.
- When you pass reference to method that parameter will referred to the same object as that referred to methods by use of call by reference

  Example:

```
class Operation2{
 int data=50;

 void change(Operation2 op){
 op.data=op.data+100;//changes will be in the instance variable
 }
  public static void main(String args[]){
  Operation2 op=new Operation2();

   System.out.println("before change "+op.data);
   op.change(op);//passing object
   System.out.println("after change "+op.data);

 }
}
```

  Output:

```
before change 50
after change 150
```

3.9.3 Difference between call by value and call by reference

No	Call by value	Call by reference
1	When a function is called and if the actual value is passes to the function then it is call by value	When function is called and if the reference or object is passed to that function then it is called call by reference
2	If the value of the parameter that are passed to the function get changed in the function body then that change is**not permanent**. That means the values of the parameters **get restored** when the control returns back to the caller function.	If the value of the parameters gets changed in the function body then that change id **permanent**. That means the value of the parameters **get changed** when the control returns back to the caller function.

3.10 Recursion

- Recursion is the process of defining something in terms of itself.
- The method that calls itself is said to be recursive.

Example:

- The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N. for example, 3 factorial is 1×2×3, or 6. Here is how a factorial can be computed by use of a recursive method.

```
class   Factorial   {
    int fact(int n) {
        int result;
    if ( n ==1) return 1;
    result = fact (n-1) * n;
    return result;
    }
}
class Recursion {
    public static void main (String args[]) {
        Factorial f =new Factorial();
        System.out.println("Factorial of 3 is " + f.fact(3));
        System.out.println("Factorial of 3 is " + f.fact(4));
        System.out.println("Factorial of 3 is " + f.fact(5));
    }
}
```

output:

```
    Factorial of 3 is 6
    Factorial of 4 is 24
    Factorial of 5 is 120
```

- When a method call itself, new local variables and parameters are allocated storage on this stack and method code is executed with these new variables from the start.
- A recursive call doesn't   not make a new copy of the method as each recursive will return the old local variable and parameter are removed from the stack and execution resume at the point of the call inside the method.
- The main advantage to recursive method is that they can be use to create clear and simpler version of several algorithms that can their interactive relatives.

Example of recursive method contain an array

```
class RecursionTest
{
            int values[];
            RecursionTest(int i)
            {
             values=new int[i];
            }
            void printArray(int i)
            {
             if(i==0)
             return;
             else
             printArray(i-1);
             System.out.println("["+(i-1)+"]"+values[i]);
            }
}
class RecursionDemo
{
            public static void main(String args[])
            {
             RecursionTest obj=new RecursionTest(10);
             int i;
             for(i=0;i<10;i++)
             obj.values[i]=i;
             obj.printArray(10);
            }
}
```

```
Output:
E:\java>javac RecursionDemo.java

E:\java>java RecursionDemo
[0]1
[1]2
[2]3
[3]4
[4]5
[5]6
[6]7
[7]8
[8]9
```

3.11 Returning object

A method can return an type of data including class type that you create.
For example, in following program increase by 10 return an object in which the value of a its invoking object

Example:

```
class Rectangle {
     int length;
     int breadth;

     Rectangle(int l,int b) {
        length = l;
        breadth = b;
      }

      Rectangle getRectangleObject() {
        Rectangle rect = new Rectangle(10,20);
        return rect;
      }
}

class Returning_Object {
      public static void main(String args[]) {
        Rectangle ob1 = new Rectangle(40,50);
        Rectangle ob2;

        ob2 = ob1.getRectangleObject();
        System.out.println("ob1.length : " + ob1.length);
        System.out.println("ob1.breadth: " + ob1.breadth);

        System.out.println("ob2.length : " + ob2.length);
        System.out.println("ob2.breadth: " + ob2.breadth);

        }
}
```

Output:

```
E:\java>set path=c:\jdk1.4.2\bin
E:\java>javac Returning_Object.java
E:\java>java Returning_Object
ob1.length : 40
ob1.breadth: 50
ob2.length : 10
ob2.breadth: 20
```

3.12 Nested class or inner class

- It is possible to define a class within another class. Such classes are known as nested class.
- The scope of a nested class is bounded by the scope of its enclosing class.
- For example if class B is defined within the class A, the B is known to A but not outside of class A.
- A nested class has access to the members including private members of the class in which it is nested.
- There are two type of nested class.

1. Static
2. Non-static

Static nested class:

- A static nested class is one which has the static modifiers apply.
- Because it is static it must access the members of its enclosing class through an objects.
- That is it can not refer to members of its enclosing class directly.
- The most important type nested class is the inner class and inner class is a non-static nested class
- Non-Static nested class :
- It has access to all of the variables and methods of its outer class and it may refer to them directly in the same way that non-static members of outer class do.
- An inner class is fully within the scope of enclosing class.

Example:

```
class outer
{
 int outer_x=100;
 void test()
 {
  inner i =new inner();
  i.display();
 }
 class inner
 {
  void display()
  {
        System.out.println("Display outer_x is: "+outer_x);
  }
 }
}
class NestedClassDemo
{
        public static void main(String args[])
        {
         outer obj=new outer();
         obj.test();
        }
}
```

Output:

```
E:\java>javac NestedClassDemo.java

E:\java>java NestedClassDemo
Display outer_x is: 100
```

3.13 Class modifiers

There are three modifiers that me precede the class keyword.

Keyword	Description
Abstract	It cannot be instantiated
Final	It cannot be extended
Public	It accessed by any other class, if **this** keyword is missing access to the class is limited to the current package.

3.13.1 Abstract class:

- Abstract classes are used to declare functionality that is implemented by one or more subclasses.
- The method in an abstract class often have empty bodies, some of the method themselves may be declared as abstract.
- There are multiple subclasses of an abstract class. These are often concrete classes that implement the functionality declared by abstract class.

3.13.2 Final class

- Final classes are sometimes declare in the manner so the methods implemented by final class cannot be overridden
- If a class is final all of its methods are also final.

3.13.3 Public class

- Public classes can be access by any other class in an application.
- If public modifier is missing the class may only be access by code in the same package.
- If none of these modifier is specified the class is assumed to be neither abstract nor final.

3.14 Inheritance

Inheritance in Java allows you to reuse code from an existing class into another class, you can derive your new class from an existing class. Your new class is called derived class which inherits all the members from its superclass.

The inherited fields can be used directly, just like any other fields. You can declare a field in the subclass with the same name as the one in the superclass, thus hiding it (not recommended). You can declare new fields in the subclass that are not in the superclass. The inherited methods can be used directly as they are. You can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it. You can write a new static method in the subclass that has the same signature as the one in the superclass, thus hiding it. You can declare new methods in the subclass that are not in the superclass. You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super. A subclass does not inherit the private members of its parent class.

```
class Shape {
  public double area ()
  {
      return 0; // Since this is just a generic "Shape" we will assume the area as zero.
              // The actual area of a shape must be overridden by a subclass, as we see below.
          // You will learn later that there is a way to force a subclass to override a method,
              // but for this simple example we will ignore that.
  }
}
class Circle extends Shape {                  // class declaration
  Circle (double diameter) {                // constructor
      this.diameter = diameter;
  }
  private static final double PI = Math.PI; // constant
  private double diameter;                  // instance variable

  public double area () {                   // dynamic method
      double radius = diameter / 2.0;
      return PI * radius * radius;
  }
}
class Rectangle extends Shape {
  // Your code goes here
}
public class Main {
  public static void main(String[] args) {
      Shape s1 = new Circle (2.5);
      Shape s2 = new Rectangle (5.0, 4.0);
      System.out.println (s1.area());
      System.out.println (s2.area());
  }
}
```

Write a program for simple inheritance

```
            class A
{
             int i,j;
             void showij()
             {
                        System.out.println("i and j is :"+i+","+j);
             }
}
class B extends A
{
            int k;
            void showk()
             {
                        System.out.println("k is :"+k);
```

```
	}
	void sum()
	{
		System.out.println("sum is :"+(i+j+k));
	}
}
class simpleInheritance
{
 public static void main(String args[])
 {
	A superob=new A();
	B subob=new B();
	superob.i=10;
	superob.j=20;
	System.out.println("Content of super class");
	superob.showij();
	subob.i=5;
	subob.j=10;
	subob.k=30;
	System.out.println("Content of sub class");
	subob.showij();
	subob.showk();
	System.out.println("sum of i,j and k is");
	subob.sum();
	}
}
D:\JAVA>javac simpleInheritance.java

D:\JAVA>java simpleInheritance
Content of super class
i and j is :10,20
Content of sub class
i and j is :5,10
k is :30
sum of i,j and k is
sum is :45
```

<u>3.15 Member access and inheritance</u>

- Although a sub class includes all of the member of it super class, it can not access those member of the super class that have been declared as private
- In a class hierarchy private member remain to their class.
- Super class variable can reference a subclass object

- A reference variable of a super class can be assign a reference to any sub class derived from that super class.
- When a sublclass become super class for another subclass this is called multilevel inheritance
- You can build hierarchies that contain as many layers of inheritant as you like
- No class can be super class of itself you can only specify one super class for any subclass that you create
- JAVA does not support multiple inheritance
- The general form of a class declaration that inherit the super class is as below

```
class subclass_name1 extends superclass_name
{
        //body of class
}
class subclass_name2 extends superclass_name
{
        //body of class
}
```

<u>3.15.1</u> <u>Super method and super keyword</u>

It has two general form

1. The first form is call the super class constructor
2. The second form is use to access a member of the super class that has been hidden by a member of sub class by the same name in the super class.

1. The first form is call the super class constructor

Syntax: super(parameterlist);

- When a subclass needs to refer to its imidiate super class, it can do so by use of the keyword super
- Super() always refered to the super class immediately above the calling class. This is true even in a multilevel hierarchy
- Super() must always be the first statement executed inside sub class constructure
- Since constructor can be overloaded super() can be called using any form defined by the super class
- The constructor executed will be the one that matches the argument
- When a class hierarchy is created in what order the constructor for the class called
- In a class hierarchy constructor are called in order of derivation from super class to subclass.
- Further super() constructor must be the first statement executed in an subclass constructor

```
class SuperClass
{
  public SuperClass(String str)
  {
    System.out.println("Super Class Constructor " + str);
  }
}
class SubClass extends SuperClass
{
  public SubClass(String str)
  {
    super(str); //calls SuperClass class constructor
    System.out.println("Sub Class Constructor " + str);
  }
}
class SuperConst
{
  public static void main(String args[])
  {
    SubClass obj = new SubClass("called");
  }
}
```

Output:

```
D:\JAVA>javac SuperConst.java
D:\JAVA>java SuperConst
Super Class Constructor called
Sub Class Constructor called
```

Here, subclass constructor uses the super() to invoke the constructor method of the super class

The following conditions are required for super method

1. Super may only be used within a subclass constructor method
2. The call to super class constructor must appear as the first statement in sub class constructor
3. The parameter in the super class must match the order and type of the instance variable declared in the super class.

2. Super Keyword

The second form is use to access a member of the super class that has been hidden by a member of sub class by the same name in the super class.

It act like accept any always refered to the super class of the subclass in which it is used.

Syntax: super.datamember;

Here member can be either a method or an instant variable

```
class SuperClass
{
  void show()
  {
  System.out.println("Super Class method has been called");
  }
}
class SubClass extends SuperClass
{
  void show()
  {
    super.show();
    System.out.println("Sub Class method has been called");
  }
}
class SuperKeyword
{
  public static void main(String args[])
 {
   SubClass obj = new SubClass();
   obj.show();
 }
}
```

Output:

```
D:\JAVA>javac SuperKeyword.java

D:\JAVA>java SuperKeyword
Super Class method has been called
Sub Class method has been called
```

3.16 Method Overriding:

When a method in a subclass has the same name and type signature as a method in its super class, than the method in super class is said to be override the method in sub class.

When an overridden method is called from within a subclass, it will always refer to the version of that method defined by this sub class

The version of the method defined by the super class will be hidden

If you wish to access the super class version of an overridden methods you can do so by using super keyword.

Method overriding occurs only when the name and the type signature of the two methods are identical.

If type signatures are not matched then its method over loading.

Rules for Java Method Overriding

1. Method must have same name as in the parent class
2. Method must have same parameter as in the parent class.
3. Method must be IS-A relationship (inheritance).

```
class Vehicle
{
        void run(){System.out.println("Vehicle is running");}
        }
```

```
class Bike2 extends Vehicle
{
 void run(){System.out.println("Bike is running safely");
      }
Class Bike3 extends Bike2
{
      void run(){System.out.println("Bike3 is in service ");

}
public static void main(String args[])
{
 Bike2 obj = new Bike2();
 obj.run();
 Bike3 ob=new Bike3();
 ob.run();
}
```

Output:

```
Bike is running safely
```

3.17 Runtime Polymorphism or Dynamic method dispatch

The dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime rather than compile time

In java it select the appropriate version of an overridden method to execute based on the class of the executing object, not the type of variable, that reference that same object

Important principles

1. A super class reference variable can refer to a subclass object. Java use this fact to resolve call to overridden method at run time
2. So when an overridden is through a super class reference java determine which version of that method to execute based upon the type of the object being referred to at the time the call occurs
3. When different type of object are referred to that it determines which version of an overridden method will executed

This is how java implements Runtime Polymorphism

```
      class A {
        void callme() {
          System.out.println("Inside A's callme method");
        }
      }
      class B extends A {
        void callme() {
          System.out.println("Inside B's callme method");
        }
      }
      class C extends A {
```

```
  void callme() {
    System.out.println("Inside C's callme method");
  }
}
class Dispatch {
  public static void main(String args[]) {
    A a = new A(); // object of type A
    B b = new B(); // object of type B
    C c = new C(); // object of type C
    A r; // obtain a reference of type A

    r = a; // r refers to an A object
    r.callme(); // calls A's version of callme

    r = b; // r refers to a B object
    r.callme(); // calls B's version of callme

    r = c; // r refers to a C object
    r.callme(); // calls C's version of callme
  }
}
D:\JAVA>javac Dispatch.java
D:\JAVA>java Dispatch
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

3.18 Advantages and Disadvantages of Inheritance

3.18.1 Advantages

1. Reusability: All public member of base class can be reused in its subclass without defining all data member again.
2. Data hiding : Base class can be define some private data members that can be hidden from its subclass
3. Overriding: With inheritance, we will be able to override the methods of the base class so that meaningful implementation of the base class method can be designed in the derived class.

3.18.2 Disadvantages

1. One of the main disadvantages of inheritance in Java is that it takes more time/effort it takes the program to redirect through all the levels of overloaded classes. If a given class has ten levels of abstraction above it, then it will essentially take ten jumps to run througha function defined in each of those classes

2. Main disadvantage of using inheritance is that the two classes (base and inherited class) get tightly coupled. This means one cannot be used independent of each other.
3. Also with time, during maintenance adding new features both base as well as derived classes are required to be changed. If a method signature is changed then we will be affected in both cases (inheritance & composition)
4. If a method is deleted in the "super class" or aggregate, then we will have to re-factor in case of using that method.Here things can get a bit complicated in case of inheritance because our programs will still compile, but the methods of the subclass will no longer be overriding superclass methods. These methods will become independent methods in their own right.

3.19 Exercise

1. Write short note : Access Protection in java Or Discuss different levels of Access Protection available in java (explain : public, private ,protected ,default)
2. Write short note on - this keyword
3. What is the difference between a static and a non-static inner class?
4. Write short notes on the following
    a. this & super keyword
    b. final & static keyword
    c. nested & inner classes & their uses.
    d. class & object .
5. Describe Dynamic method dispatch concept. Also give the difference between Abstract class & interface.
6. What are the conditions for using super() method.
7. Define constructor. How do we invoke constructor in JAVA?
8. Define method overloading.
9. Write various types of inheritance.
10. Define class. How does it accomplish data hiding?
11. Explain super and final keyword with example.

4. INTERFACE and PACKAGE

4.1 Interface

Interface is something to class which defines what to do but not how to do. Interface is syntactically similar to classes but they lack instance variable and their methods are declared without any body. Once the interface is define, any number of classes can implement an interface & one class can implement any number of interface.
To implement an interface a class must create a complete set of methods defined by the interface. However each class is free to determine the details of its own implementation. By providing the 'interface' keyword, java allows you to have "1 interface multiple methods" which is called as property polymorphism.
Syntax:

```
Accesss specifier interface  interface_name
{
final/static datatype variablename1 = value ;
final/static datatype variablename2 = value ;
:
:
datatype method1 ([argument list]);
datatype method2 ([argument list]);
:
:
}
```

Example:

```
public interface bright
{
        final int max_tv = 50;
        final int min_tv = 20;
        void increase(int x);
        void decrease(int x);
}
```

1) Here in the place of access specifier, you can either use public or no access specifier. If you declare the interface with the public access specifier, you can use this interface within package or in any other package but without access specifier the interface which is declared can be use only within that package.
2) Interface name specify any valid interface name.
3) The variables declared in the interface will be either static or final meaning they can’t be change by the implementing class & they must also initialize with some constant value.
4) The methods which are declared in the interface have no bodies. They just ends with semicolon after parameter list. They are essentially abstract method that means they just have the signature of the method but no codes within body.

An Interfaces have following properties

1. An interface is implicitly abstract. You do not need to use the abstract keyword when declaring an interface.
2. Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
3. Methods in an interface are implicitly public.
4. An interface is similar to a class in the following ways:
5. An interface can contain any number of methods.
6. An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.
7. The bytecode of an interface appears in a .class file.
8. Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.
9. However, an interface is different from a class in several ways, including:
10. An interface does not contain any constructors.
11. All of the methods in an interface are abstract.
12. An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
13. An interface is not extended by a class; it is implemented by a class.
14. An interface can extend multiple interfaces.

NOTE: All methods declared in an interface are implicitly public so the public modifier can be omitted.

4.2 Implementing an interface:

Once an interface has been defined, one or more classes implement that interface. To implement the interface, include "**implements**" clause in the class definition.
Syntax:
Class_name implements interface1, interface2,……..
The methods (of interface) which are implemented by class should be declared as public during overriding although you don't have specified those methods as public during interface declaration.

4.3 Accessing the implementation through interface reference:

This kind of accessing is just because during the dynamic method dispatch or dynamic runtime polymorphism the compiler needs to have a reference of the super class where as in this case interface reference use for dynamic binding.
Syntax:
Interfacename object = new classname([argument list]);
Ex: bright t= new tv();
Where bright is an interface and tv is class.
Program:

```
interface MyInterface
{
  public void method1();
  public void method2();
}
class XYZ implements MyInterface
{
 public void method1()
 {
    System.out.println("implementation of method1");
 }
 public void method2()
 {
    System.out.println("implementation of method2");
 }
 public static void main(String arg[])
 {
    MyInterface obj = new XYZ();
    obj. method1();
 }
}
```

Output:

```
implementation of method1
```

4.4 Interface Extension:

One interface can inherit another interface by the use of keyword “extends”. Syntax is same as for inheriting classes. When a class implements an interface that inherits another interface. It must provide implementation for all the methods define within the interface chain.

Program:

```
public interface I1
{
void method1();
}
public interface I2 extends I1
{
  void method2();
}
class Demo implements I2
{
 public void method1()
{
       System.out.println("Interface1’s method1");
 }
 public void method2()
{
    System.out.println("Interface2’s method2");
 }
};//Demo
public class IntExt
{
public static void main(String arg[])
 {
    Demo d = new Demo();
    d. method1();
    d. method2();
 }
};//IntExt
```

In above case interface I2 inherits properties of interface I1. Although class Demo implements the interface I2, it should override all the methods declared in interface I2 and I1 both.

4.5 Partial Implementation:

If a class includes an interface but does not fully implement the methods defined by the interface then it is called as partial implementation. The class which uses the partial implementation should be declared as abstract class and the subclass of an abstract class must override the methods which are not overridden in the abstract class.

Program:

```
public interface I1
{
void method1();
void method2();
}
abstract class Demo implements I1
{
  public void method1()
{
        System.out.println("method1 displayed");
  }
 };//Demo

public class IntExt extends Demo
{
public void method2()
{
        System.out.println("method2 displayed");
  }
public static void main(String arg[])
  {
     Demo d = new Demo();
     d. method1();
     d. method2();
  }
};//IntExt
```

In above case interface I1 implemented by class Demo but it only override method1. So Demo class declared as an abstract class and method2 which is not overridden by class Demo must be overridden in subclass of Demo class i.e. IntExt.

4.6 Interface and multiple inheritance

1. Interface has another important role in the Java programming language. Interfaces are not part of the hierarchy although they work in combination with classes.
2. The Java programming language does not permit multiple inheritances but interface provide an alternative to these.
3. In Java a class can inherit from only one class, but it can implements more than one interface.
4. There for object can have multiple types, the type of their own class and the type of all the interfaces that they implements.
5. These means that if variable is declared to be the type of an interface its value can reference any object that is instantiated from any class that implements the interface.

Example:

```
import java.lang.*;
import java.io.*;
interface Exam
{
        void percent_cal();
}
class Student
{
        String name;
        int roll_no,mark1,mark2;
        Student(String n, int r, int m1, int m2)
        {
                name=n;
                roll_no=r;
                mark1=m1;
                mark2=m2;
        }
        void display()
        {
                System.out.println ("Name of Student: "+name);
                System.out.println ("Roll No. of Student: "+roll_no);
                System.out.println ("Marks of Subject 1: "+mark1);
                System.out.println ("Marks of Subject 2: "+mark2);
        }
}
class Result extends Student implements Exam
{
        Result(String n, int r, int m1, int m2)
        {
                super(n,r,m1,m2);
        }
        public void percent_cal()
        {
                int total=(mark1+mark2);
                float percent=total*100/200;
                System.out.println ("Percentage: "+percent+"%");
        }
        void display()
        {
                super.display();
        }
}
class Multiple_Inheritance
```

```
{
        public static void main(String args[])
        {
                Result R = new Result("Ra.one",12,93,84);
                R.display();
                R.percent_cal();
        }
}
```

Output:

```
D:\javap>javac Multiple_Inheritance.java
D:\javap>java Multiple_Inheritance
Name of Student: Ra.one
Roll No. of Student: 12
Marks of Subject 1: 93
Marks of Subject 2: 84
Percentage: 88.0%
```

4.7 Differences

4.7.1 Differences: abstract class vs interface

Abstract class	**Interface**
1) Abstract class can **have abstract and non-abstract**methods.	Interface can have **only abstract** methods.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can have static methods, main method and constructor**.	Interface **can't have static methods, main method or constructor**.
5) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
6) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

7) **Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Note: Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

<u>4.7.2</u> <u>Difference: Concrete class vs. Abstract class vs. Interface</u>

1. A concrete class has concrete methods, i.e., with code and other functionality. This class a may extend an abstract class or implements an interface.
2. An abstract class must contain at least one abstract method with zero or more concrete methods
3. An interface must contain only method signatures and static members.
4. An interface does not have definitions of the methods declared in it.
5. An abstract class may have some definition and at least one abstract method.
6. A subclass of an abstract class must either implement all the abstract methods of the abstract class or declare itself as abstract.

Key points: Here are the key points to remember about interfaces:
1) We can't instantiate an interface in java.
2) Interface provides complete abstraction as none of its methods can have body. On the other hand, abstract class provides partial abstraction as it can have abstract and concrete (methods with body) methods both.
3) implements keyword is used by classes to implement an interface.
4) While providing implementation in class of any method of an interface, it needs to be mentioned as public.
5) Class implementing any interface must implement all the methods, otherwise the class should be declared as "abstract".
6) Interface cannot be declared as private, protected or transient.
7) All the interface methods are by default abstract and public.
8) Variables declared in interface are public, static and final by default.

```
interface Try
{
  int a=10;
  public int a=10;
  public static final int a=10;
  final int a=10;
  static int a=0;
}
```

All of the above statements are identical.
9) Interface variables must be initialized at the time of declaration otherwise compiler will through an error.

```
interface Try
{
    int x;//Compile-time error
}
```

Above code will throw a compile time error as the value of the variable x is not initialized at the time of declaration.

10) Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final. Here we are implementing the interface “Try” which has a variable x. When we tried to set the value for variable x we got compilation error as the variable x is public static final by default and final variables can not be re-initialized.

```
Class Sample implements Try
{
  public static void main(String arg[])
  {
     x=20; //compile time error
  }
}
```

11) Any interface can extend any interface but cannot implement it. Class implements interface and interface extends interface.

12) A class can implement any number of interfaces.

13) If there are two or more same methods in two interfaces and a class implements both interfaces, implementation of the method once is enough.

```
interface A
{
   public void aaa();
}
interface B
{
   public void aaa();
}
class Central implements A,B
{
   public void aaa()
   {
       //Any Code here
   }
   public static void main(String arg[])
   {
       //Statements
    }
}
```

14) A class cannot implement two interfaces that have methods with same name but different return type.

```
interface A
{
   public void aaa();
}
interface B
{
   public int aaa();
}
class Central implements A,B
{

   public void aaa() // error
   {
   }
   public int aaa() // error
   {
   }
   public static void main(String arg[])
   {

   }
}
15) Variable names conflicts can be resolved by interface name
interface A
{
   int x=10;
}
interface B
{
   int x=100;
}
class Hello implement A,B
{
   public static void Main(String arg[])
   {

      System.out.println(x); // reference to x is ambiguous both variables are x
      System.out.println(A.x);
      System.out.println(B.x);
   }
}
```

4.8 Advantages and Disadvantages of interfaces:

Advantages of interfaces:

1. Without bothering about the implementation part, we can achieve the security of implementation
2. In java, **multiple inheritance** is not allowed, However by using interfaces you can achieve the same. A class can extend only one class but can implement any number of interfaces.

3. Interface provides a contract for all the implementation classes, so it's good to code in terms of interfaces because implementation classes can't remove the methods we are using.
4. Interfaces are good for starting point to define Type and create top level hierarchy in our code.
5. Since a java class can implements multiple interfaces, it's better to use interfaces as super class in most of the cases.

Disadvantages of interfaces:
1. We need to choose interface methods very carefully at the time of designing our project because we can't add or remove any methods from the interface at later point of time, it will lead compilation error for all the implementation classes. Sometimes this leads to have a lot of interfaces extending the base interface in our code that becomes hard to maintain.
2. If the implementation class has its own methods, we can't use them directly in our code because the type of Object is an interface that doesn't have those methods. For example, in above code we will get compilation error for code shape.getRadius(). To overcome this, we can use **typecasting** and use the method like this:
Circle c = (Circle) shape;
c.getRadius();
Although class typecasting has its own disadvantages.

4.9 PACKAGE INTRODUCTION

Files in one directory (or package) would have special functionality from those of another directory. For example, files in java.io package do something related to I/O, but files in java.net package give us the way to deal with the Network, meaning that this directory keeps files related to the presentation part of the application. Packaging also helps us to avoid class name collision when we use the same class name as that of others.
Simple package designed to include user interactivity. The package enables beginner programmers to build programs with GUI-like interactivity while maintaining good design principles. An advantage of this package is that it is easy to implement using the classes. Therefore, it can be used as a case study to demonstrate Java features.

4.10 How to create a package
Suppose we have a file called HelloWorld.java, and we want to put this file in a package **world**. First thing we have to do is to specify the keyword **package** with the name of the package we want to use (**world**in our case) on top of our source file, before the code that defines the real classes in the package, as shown in our HelloWorld class below:

```
// only comment can be here
package world;

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

```
        }
      }
```

One thing you must do after creating a package for the class is to create nested subdirectories to represent package hierachy of the class. In our case, we have the **world** package, which requires only one directory. So, we create a directory **world** and put our HelloWorld.java into it.

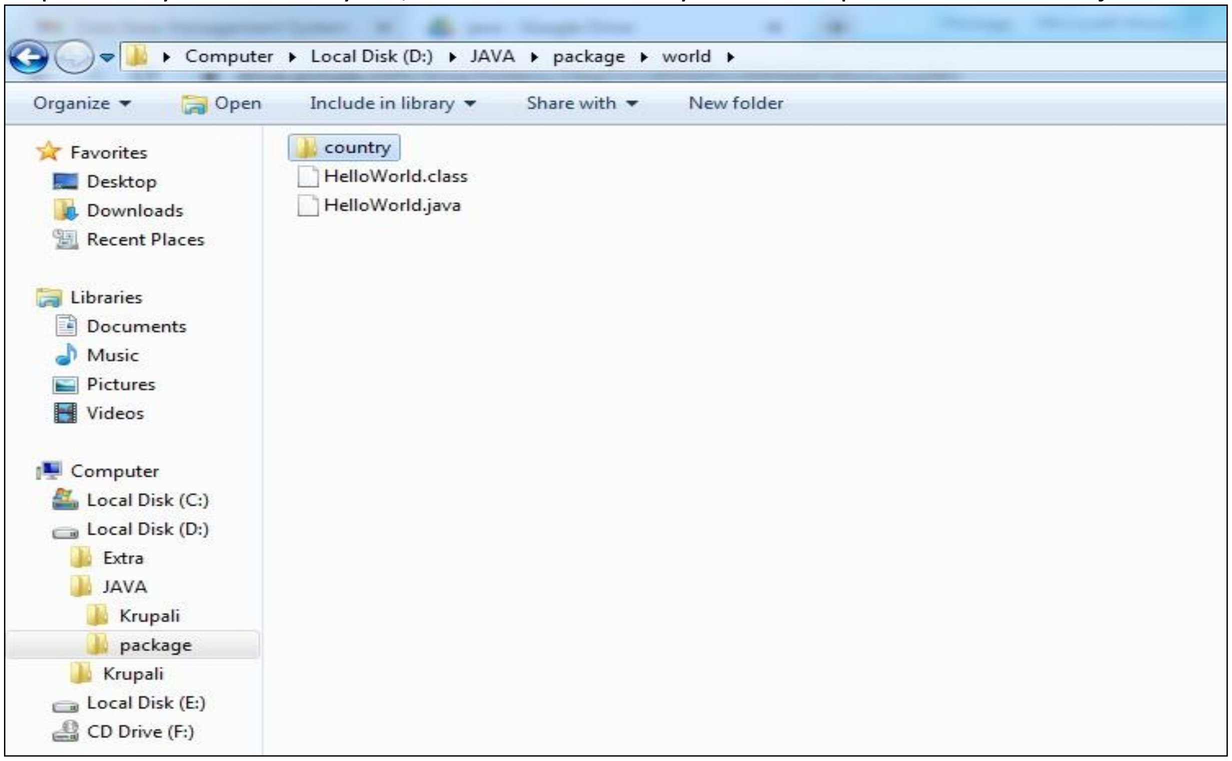


Figure: **HelloWorld** in world package (**D:\JAVA\package\world**)

That's it!!! Right now we have HelloWorld class inside world package. Next, we have to introduce the world package into our **CLASSPATH**.

<u>4.10.1</u> <u>Setting up the CLASSPATH</u>

From figure 2 we put the package world under d:. So we just set our CLASSPATH as:
set CLASSPATH=.;D:\;
We set the CLASSPATH to point to 2 places, . (dot) and C:\ directory.

Note: If you used to play around with DOS or UNIX, you may be familiar with . (dot) and .. (dot dot). We use . as an alias for the current directory and .. for the parent directory. In our CLASSPATH we include this . for convenient reason. Java will find our class file not only from C: directory but from the current directory as well. Also, we use ; (semicolon) to separate the directory location in case we keep class files in many places.

When compiling HelloWorld class, we just go to the world directory and type the command:
D:\JAVA\package\world \javac HelloWorld.java
If you try to run this HelloWorld using **java HelloWorld**, you will get the following error:
D:\JAVA\package\world >java HelloWorld
Exception in thread "main" java.lang.NoClassDefFoundError: HelloWorld (wrong name:

```
world/HelloWorld)
        at java.lang.ClassLoader.defineClass0(Native Method)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:442)
        at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:101)
        at java.net.URLClassLoader.defineClass(URLClassLoader.java:248)
        at java.net.URLClassLoader.access$1(URLClassLoader.java:216)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:197)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:191)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:290)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:286)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:247)
```

The reason is right now the HelloWorld class belongs to the package world. If we want to run it, we have to tell JVM about its **fully-qualified class name** (world.HelloWorld) instead of its plain class name(HelloWorld).

```
D:\JAVA\package\world >java world.HelloWorld
D:\JAVA\package\world >Hello World
```

Note: **fully-qualified class name** is the name of the java class that includes its package name

4.10.2 Subpackage (package inside another package)

Assume we have another file called **HelloCountry.java**. We want to store it in a subpackage **"country"**, which stays inside package **world**. The HelloCountry class should look something like this:

```
package world.country;

public class HelloCountry {
  private String Name = "Jump to Learn";
  public getName() {
    return Name;
  }
  public setNAme(String SName) {
    this. SName = SName;
  }
}
```

If we store the package world under C: as before, the **HelloCountry.java** would be **D:\JAVA\package \world\country\ HelloCountry.java** as shown in Figure 4 below:

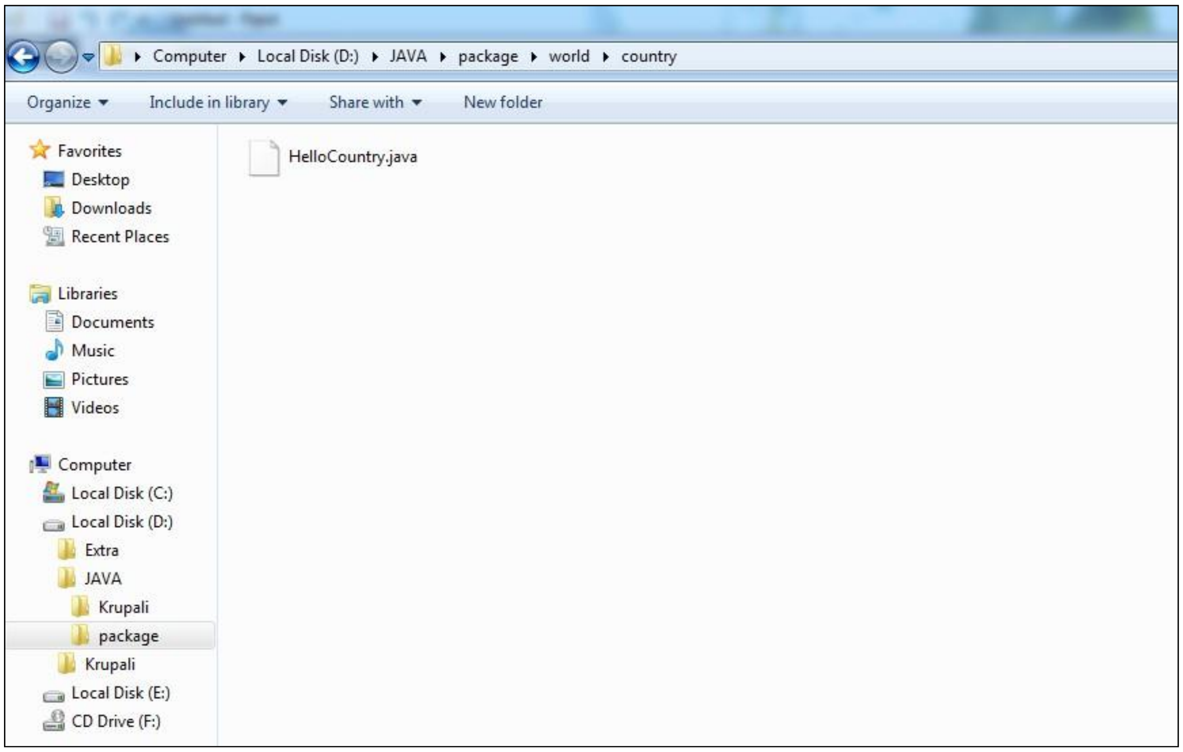


Figure: **HelloCountry** in **world.country** package

Although we add a subpackage under package world, we still don't have to change anything in our CLASSPATH. However, when we want to reference to the HelloCountry class, we have to use world.country.HelloCountry as its fully-qualified class name.

<u>4.11</u> <u>How to use package</u>

There are 2 ways in order to use the public classes stored in package.

1. Declare the fully-qualified class name. For example,

/*

world.HelloWorld HW = new world.HelloWorld();

world.country. HelloCountry HC = new world.country.HelloCountry ();

String Name = HC.getName();

*/

2) Use an "import" keyword:

import world.*; // we can call any public classes inside the world package

import world.country.*; // we can call any public classes inside the world.country package

import java.util.*; // import all public classes from java.util package

import java.util.Hashtable; // import only Hashtable class (not all classes in java.util package)

Thus, the code that we use to call the HelloWorld and HelloCountry class should be

/*

HelloWorld HW = new HelloWorld(); // don't have to explicitly specify world.HelloWorld anymore

HelloCountry HC = new HelloCountry (); // don't have to explicitly specify world.country. HelloCountry anymore
*/
Note that we can call public classes stored in the package level we do the import only. We can't use any classes that belong to the subpackage of the package we import. For example, if we import package world, we can use only the HelloWorld class, but not the HelloCountry class.

4.12 Characteristics of packages

Packages are stored in hierarchical manner and are explicitly imported into new classes definition.
The package is both naming and visibility control mechanism.
If a source file doesn't contain a package statement, its classes and interfaces are placed in a default package.
Package statement must appear as the first statement.
Two different packages can declare classes have same, by providing this way JAVA provides way for partitioning the class name space into more manageable chunks.
We can define classes inside package that are not accessible by code outside that package.
We can also define class members that are only exposed to other members of the same package.

4.13 Benefits of Package

The classes contained in the packages of other programs can be easily reused.
In packages, classes can be unique compared with classes in other packages. i.e. two classes in two different packages can have same name.
Packages can hide classes that are meant to internal use only from other programs or packages.
Packages also provide a way for separating "design" and "coding". 1st we can design classes and decide their responsibility and then we can implement the java code needed for the methods. It is possible to change the implementation of any method without affecting the rest of the design.

4.14 Hierarchy of packages

One can create hierarchy of packages. To do this simply separate each package name from the one above it by use of a period.
Syntax: package pkg1[.pkg2[.pkg3]];
CLASSPATH variable
CLASSPATH is an environment variable that determines where the JDK tools such as the JAVA compiler and interpreter search for a .class file. It contains an ordered sequence of directories as well as .jar and .zip files. It can be set as:
set classpath=.;c:\\project1;c:\\project2
echo %classpath% displays the current settings of the classpath variable.
Example:

```
package MyPack;
class Balance
{
    String name;
    double bal;
    Balance(String n, double b)
    {
            name=n;
            bal=b;
    }
    void show()
    {
            If(bal<0)
            {
                    System.out.println("->  ");
                    System.out.println(name+":Rs."+bal);
            }
    }
}
class AccountBalance
{
    public static void main(String args[])
    {
            Balance current[]=new Balance[5];
            Current[0]=new Balance["liza",25000.02];
            Current[1]=new Balance["liza",17000.00];
Current[2]=new Balance["pushpendu",12000.00];
            for(i=0;i<3;i++)
            current[i].show();
    }
}
```

4.15 Package Access Modifier

As you will see, Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages. Packages act as containers for classes and other subordinate packages.

Java addresses four categories of visibility for class members:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

The three access specifiers, private, public, and protected, provide a variety of ways to produce the many levels of access required by these categories.

CLASS MEMBER ACCESS

No.	PRIVATE	NO MODIFIER	PROTECTED	PUBLIC
SAME CLASS	YES	YES	YES	YES
SAME PACKAGE SUBCLASS	NO	YES	YES	YES
SAME PACKAGE NON-SUBCLASS	NO	YES	YES	YES
DIFFERENT PACKAGE SUBCLASS	NO	NO	YES	YES
DIFFERENT PACKAGE NON-SUBCLASS	NO	NO	NO	YES

1. **PUBLIC**
A class has only two possible access levels: default and public.
When a class is declared as public, it is accessible by any other code.
DEFAULT [friendly access]
If a class has default access, then it can only be accessed by other code within its same package.
The difference between public access and friendly access is that the public modifier makes fields visible in all classes regardless of their packages, while friendly access modifier makes field's visibility only in the same package but not in other packages.

2. **PRIVATE**
Anything declared private cannot be seen outside of its class.
It can't be inherited by the subclasses of other packages.

3. **PROTECTED**
Protected modifier makes the field visible not only to all classes and subclasses in the same package as well as in other packages.

4. **PRIVATE PROTECTED**
This modifier makes the fields visible in all sub classes regardless of what package they are in. These fields are not accessible by other classes in the same package.

EXAMPLE
The following example shows all combinations of the access control modifiers.
Following is the source code for the other package, p1.

```
//This is file Protection.java:
package p1;
public class Protection
{
     int n = 1;
     private int n_pri = 2;
    protected int n_pro = 3;
    public int n_pub = 4;
    public Protection()
    {
     System.out.println("base constructor");
     System.out.println("n = " + n);
     System.out.println("n_pri = " + n_pri);
     System.out.println("n_pro = " + n_pro);
     System.out.println("n_pub = " + n_pub);
     }
}

//This is file Derived.java:
package p1;
class Derived extends Protection
{
     Derived()
     {
             System.out.println("derived constructor");
             System.out.println("n = " + n);
             // class only
             // System.out.println("n_pri = " + n_pri);
             System.out.println("n_pro = " + n_pro);
             System.out.println("n_pub = " + n_pub);
     }
}
This is file SamePackage.java:
package p1;
class SamePackage
{
SamePackage()
 {
Protection p = new Protection();
System.out.println("same package constructor");
System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

Following is the source code for the other package, p2.

```
//This is file Protection2.java:
package p2;
class Protection2 extends p1.Protection
{
    Protection2()
    {
    System.out.println("derived other package constructor");
    // class or package only
    // System.out.println("n = " + n);
    // class only
    // System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
    }
}

//This is file OtherPackage.java:
package p2;
class OtherPackage
 {
    OtherPackage()
    {
    p1.Protection p = new p1.Protection();
    System.out.println("other package constructor");
    // class or package only
    // System.out.println("n = " + p.n);
    // class only
    // System.out.println("n_pri = " + p.n_pri);
    // class, subclass or package only
    // System.out.println("n_pro = " + p.n_pro);
    System.out.println("n_pub = " + p.n_pub);
    }
}
```

Here are two test files you can use. The one for package p1 is shown here:

```
// Demo package p1.
package p1;
// Instantiate the various classes in p1.
public class Demo
```

```
{
    public static void main(String args[])
    {
            Protection ob1 = new Protection();
            Derived ob2 = new Derived();
            SamePackage ob3 = new SamePackage();
    }
}
```

The test file for p2 is shown next:

```
// Demo package p2.
package p2;
// Instantiate the various classes in p2.
public class Demo
 {
    public static void main(String args[])
    {
            Protection2 ob1 = new Protection2();
            OtherPackage ob2 = new OtherPackage();
    }
}
```

4.16 Exercise

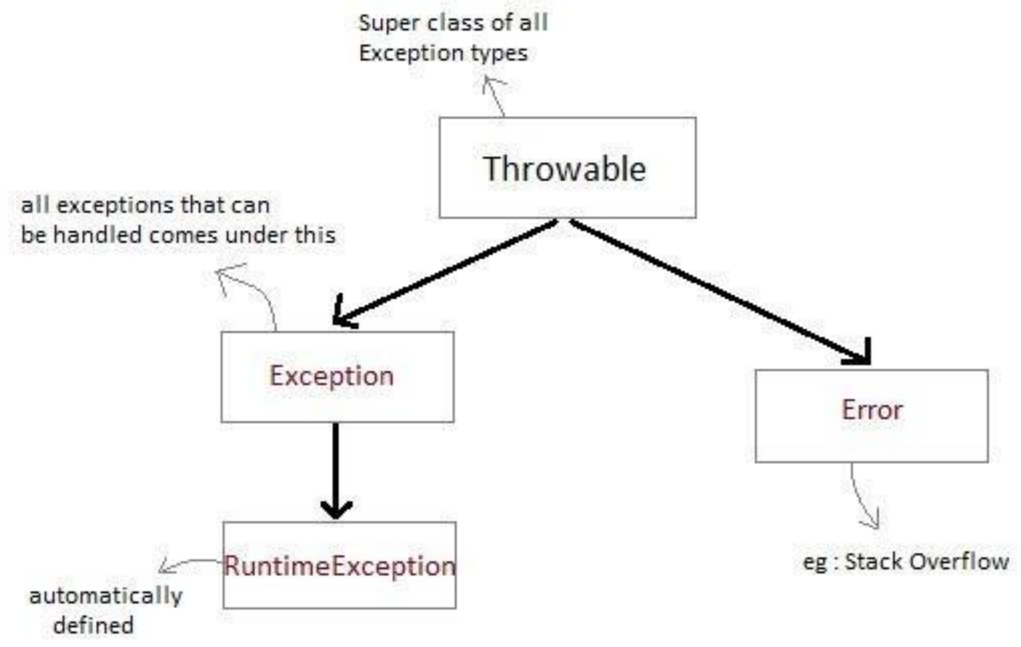
1. What is interface in java?
2. Can we achieve multiple inheritance by using interface?
3. How to declare an interface, write a syntax?
4. Can we create an object of an interface?
5. Can we declare the interface as final?
6. Which keyword java compiler add before interface fields and methods?
7. Does interface extend Object class by default?
8. Can an interface extend another interface?
9. Can an interface extend a class?
10. Can we put a static method in interfaces?
11. Can we declare an interface with the abstract keyword?
12. What is default keyword in an interface?
13. Can we declare a constructor in the interface?
14. After compilation of interface program, .class file will be generated for every interface in java. true or false.?
15. Can we change the value of a field in interface after initialization?
16. Difference between abstract class and interface?
17. Write a short note on package. Give its significance
18. Discuss various levels of access protection available for package.

19. Define package. How do we design a package and a class or an interface to a package?
20. Which keyword protects a class from accessing outside a package?
21. Where does the class files stored, when we don’t explicitly create a package?
22. What is default modifier? Give its visibility in a package?
23. What do you mean by sealed package?
24. Explain package. Why we import package?

5. Exception and Strings

5.1 Exception:

An exception (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore these exceptions are to be handled. Exception can occur at runtime (known as runtime exceptions) as well as at compile-time (known Compile-time exceptions).


Super class of all Exception types
Throwable
all exceptions that can be handled comes under this
Exception
Error
RuntimeException
automatically defined
eg : Stack Overflow

- The 'Throwable' class is the superclass of all errors and exceptions in the Java language and is at the top of exception class hierarchy.
- Subclasses of throwable class are error and exception.
- Errors indicate serious problems and abnormal conditions that most applications should not try to handle. Error defines problems that are not expected to be caught under normal circumstances by our program. For example memory error, hardware error, JVMerror etc.
- Exceptions are conditions within the code. A developer can handle such conditions and take necessary corrective actions. Few examples –
    1. DivideByZero exception
    2. NullPointerException
    3. ArithmeticException
    4. ArrayIndexOutOfBoundsException
- RuntimeException is a subclass of Exception. Exceptions under this class are automatically defined for programs.

<u>5.2</u> <u>Exception Types:</u>

An exception can occur for many different reasons, below given are some scenarios where exception occurs.

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A 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. Based on these we have categories exception mainly in **two** types

Checked and Unchecked where **Error** is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Checked Exception / Programmatic Exceptions / compile time exceptions:

1. The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException, File that need to be opened is not found, ClassNotFoundException, IllegalAccessException, NoSuchFieldException EOFException etc.
2. The exception that can be predicted by the programmer.
3. Checked exceptions are checked at these are also called as compile time exceptions.
4. Compile-time exceptions cannot simply be ignored at the time of compilation, the Programmer should take care of (handle) these exceptions.

Unchecked Exception / JVM Exceptions / Runtime Exceptions:

- Runtime Exceptions are also known as Unchecked Exceptions as the compiler do not check whether the programmer has handled them or not but it’s the duty of the programmer to handle these exceptions and provide a safe exit. In short Unchecked exceptions are checked at runtime & ignored at compile time.

- Unchecked exceptions are the class that extends RuntimeException. Which means RuntimeException is the parent class of all runtime exceptions.
- If we are throwing any runtime exception in a method, it's not required to specify them in the method signature throws clause.
- Runtime Exceptions are cause by bad programming. These include programming bugs, such as logic errors or improper use of an API. 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.
  Examples
  ArithmeticException
  ArrayIndexOutOfBoundsException
  NullPointerException
  NegativeArraySizeException etc.

Error

- Error is irrecoverable
- These 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.That's why we have a separate hierarchy of errors and we should not try to handle these situations.
- Some of the common Errors are OutOfMemoryError and StackOverflowError, hardware failure, JVM crash or out of memory error, VirtualMachineError, AssertionError etc.

Java provides a robust and object oriented way to handle exception scenarios, known as **Java Exception Handling**.

5.3 Exception handlers

In java, exception handling is done using five keywords,

1. **try**
2. **catch**
3. **throw**
4. **throws**
5. **finally**

When exception occurs, the execution of a program is transfer to an appropriate exception handler

5.3.1 Try & catch:

- The codes which have possibility to generate exceptions are placed in the try block, when an exception occurs, that exception is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.
- try is the start of the block and catch is at the end of try block to handle the exceptions.
- catch block requires a parameter that should be of type Exception.

- Code within a try/catch block is referred to as protected code,

Syntax:

```
try
{
//Protected code
}catch(ExceptionName e1)
{
//Catch block
}
```

Program:

```
class Example1 {
  public static void main(String args[]) {
    int num1, num2;
    try {
      // Try block to handle code that may cause exception
      num1 = 0;
      num2 = 62 / num1;
      System.out.println("Try block message");
    } catch (ArithmeticException e) {
        // This block is to catch divide-by-zero error
        System.out.println("Error: Don't divide a number by zero");
     }
    System.out.println("I'm out of try-catch block in Java.");
  }
}
```

Output:

```
Error: Don't divide a number by zero
I'm out of try-catch block in Java.
```

5.3.1.1. Multiple catch blocks in Java

We can have multiple catch blocks with a try and try-catch block

Syntax:

```
try
{
//Protected code
}catch(ExceptionType1 e1)
{
//Catch block
}catch(ExceptionType2 e2)
{
//Catch block
}catch(ExceptionType3 e3)
{
//Catch block
}
```

1. A try block can have any number of catch blocks.
2. A catch block that is written for catching the class Exception can catch all other exceptions

Syntax:

```
catch(Exception e){
  //This catch block catches all the exceptions
}
```

3. If multiple catch blocks are present in a program then the above mentioned catch block should be placed at the last as per the exception handling best practices.
4. If the try block is not throwing any exception, the catch block will be completely ignored and the program continues.
5. If the try block throws an exception, the appropriate catch block (if one exists) will catch it

–catch(ArithmeticException e) is a catch block that can catch ArithmeticException

–catch(NullPointerException e) is a catch block that can catch NullPointerException

6. All the statements in the catch block will be executed and then the program continues.

<u>Program:</u>

```
class Example2{
   public static void main(String args[]){
     try{
         int a[]=new int[7];
         a[4]=30/0;
         System.out.println("First print statement in try block");
     }
     catch(ArithmeticException e){
        System.out.println("Warning: ArithmeticException");
     }
     catch(ArrayIndexOutOfBoundsException e){
        System.out.println("Warning: ArrayIndexOutOfBoundsException");
     }
     catch(Exception e){
        System.out.println("Warning: Some Other exception");
     }
   System.out.println("Out of try-catch block...");
  }
}
```

Output:

```
Warning: ArithmeticException
Out of try-catch block...
```

In the above example there are multiple catch blocks and these catch blocks executes sequentially when an exception occurs in try block. Which means if you put the last catch block (catch(Exception e)) at the first place, just after try block then in case of any exception this block will execute as it has the ability to handle all exceptions. This catch block should be placedat the last to avoid such situations.

5.3.1.2 Nested try catch blocks in Java

The try catch blocks can be nested. One try-catch block can be present in the another try'sbody. This is called Nesting of try catch blocks. Each time a try block does not have a catch handler for a particular exception, the stack is unwound and the next try block's catch (i.e., parent try block's catch) handlers are inspected for a match.

If no catch block matches, then the java run-time system will handle the exception.

Syntax:

```
try
{
statement 1;
statement 2;
try
{
statement 3;
statement 4;
}
catch(Exception e)
{
//Exception Message
}
}//main try over
catch(Exception e) //Catch of Main(parent) try block
{
//Exception Message
}//main catch over
```

The main point to note here is that whenever the child try-catch blocks are not handling any exception, the control comes back to the parent try-catch if the exception is not handled there also then the program will terminate abruptly.

Program:

```
class Nest{
  public static void main(String args[]){
          //Parent try block
    try{
          //Child try block1
       try{
         System.out.println("Inside block1");
         int b =45/0;
         System.out.println(b);
       }
       catch(ArithmeticException e1){
         System.out.println("Exception: e1");
       }
       //Child try block2
```

```
        try{
          System.out.println("Inside block2");
          int b =45/0;
          System.out.println(b);
        }
        catch(ArrayIndexOutOfBoundsException e2){
          System.out.println("Exception: e2");
        }
       System.out.println("Just other statement");
   }
   catch(ArithmeticException e3){
           System.out.println("Arithmetic Exception");
       System.out.println("Inside parent try catch block");
   }
   catch(ArrayIndexOutOfBoundsException e4){
          System.out.println("ArrayIndexOutOfBoundsException");
       System.out.println("Inside parent try catch block");
   }
   catch(Exception e5){
          System.out.println("Exception");
       System.out.println("Inside parent try catch block");
    }
    System.out.println("Next statement..");
  }
}
```

Output:

```
Inside  block1
Exception: e1
Inside block2
Arithmetic Exception
Inside parent try catch block
Next statement..
```

<u>5.3.2</u> <u>Throw</u>

Throw keyword is used to throw an exception explicitly. Only object of Throwable class or its sub classes can be thrown. Program execution stops on encountering throw statement, and the closest catch statement is checked for matching type of exception. If it doesn’t find match it will go for another catch & so..on. if it doesn’t find match at all then a default handler halts the program and gives stack trace message. We can throw either checked or uncheked exceptions in java by throw keyword. The throw keyword is mainly used to throw custom exception or user defined exception.

Syntax :

```
throw Throwableobject
```

Creating object of Throwable class
There are two possible ways to get an instance of class Throwable,

- Using a parameter in catch block.
- Creating instance with new operator.

Example:
throw new IOException("sorry device error");
This constructs an instance of IOException with name sorry device error.
new NullPointerException("test");
This constructs an instance of NullPointerException with name test.

<u>Program:</u>

```
class Test
{
 static void avg()
 {
  try
  {
   int i= 5/0 throw new ArithmeticException("demo");
  }
  catch(ArithmeticException e)
  {
   System.out.println("Exception caught");
  }
 }

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

In the above example the avg() method throw an instance of ArithmeticException, which is successfully handled using the catch statement.

<u>5.3.3</u> <u>Throws</u>

Any method capable of causing exceptions must list all the exceptions possible during its execution, so that caller can guard themselves from these exceptions using exception handling codes so normal flow can be maintained.A method can do so by using the throws keyword. <u>Syntax :</u>

```
type method_name(parameter_list) throws exception_list
{
 //definition of method
}
```

NOTE: It is necessary for all exceptions, except the exceptions of type Error and Runtime-Exception or any of their subclass.

Program:

```
class Test
{
 static void check() throws ArithmeticException
 {
  System.out.println("Inside check function");
  throw new ArithmeticException("demo");
}
 public static void main(String args[])
 {
  try
  {
   check();
  }
  catch(ArithmeticException e)
  {
   System.out.println("caught" + e);
  }
 }
}
```

Advantage of throws:

Now Checked Exception can be propagated (forwarded in call stack).
It provides information to the caller of the method about the exception.

5.3.4 Difference between throw and throws in Java

No.	Throw	Throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

1)Java throw example

```
void m(){
throw new ArithmeticException("sorry");
}
```

2)Java throws example

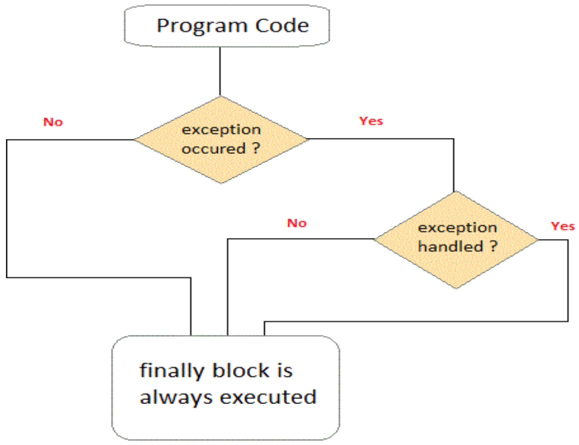
```
void m()throws ArithmeticException{
//method code
}
```

3) Java throw and throws example

```
void m()throws ArithmeticException{
throw new ArithmeticException("sorry");
}
```

5.3.5 Finally:

A finally keyword is used to create a block of code that follows a try block. A finally block ofcode always executes whether or not exception has occurred. A finally block appears at the endof catch block. We can have only one finally block for each try statement.



1. If exception occurs in try block's body then control immediately transferred (**skipping rest of the statements in try block**) to the catch block. Once catch block finishedexecution then **finally block** completes its execution and then the rest of the program.
2. It is not mandatory to include a finally **block** at all, but if you do, it will run regardless of whether an exception was thrown and handled by the try and catch blocks.
3. If there is no exception occurred in the code which is present in try block then first, the try block gets executed completely and then control gets transferred to finally block (**skipping catch blocks**).
4. If a **return statement** is encountered either in try or catch block. In such case also **finally runs**. Control first goes to finally and then it returned back to **return statement.**

Syntax:

```
try
{
  //Protected code
}

catch(ExceptionType e)
{
  //Catch block
}
: ….
finally
{
  //The finally block always executes.
}
```

Program:

```
Class ExceptionTest
{
 public static void main(String[] args)
 {
  int a[]= new int[2];
  System.out.println("out of try");
  try
  {
   System.out.println("Access invalid element"+ a[3]);
   /* the above statement will throw ArrayIndexOutOfBoundException */
  }
  finally
  {
   System.out.println("finally is always executed.");
  }
 }
}
```

Output:

```
Out of try
finally is always executed.
Exception in thread main java. Lang. exception array Index out of bound exception.
```

In above example even if exception is thrown by the program, which is not handled by catch block, still finally block will get executed.

5.4 User-defined Exceptions:

You can create your own exceptions in Java by creating your own exception sub class simply by extending java Exception class. You can define a constructor for your Exception sub class (not compulsory) and you can override the toString() function to display your customized message on catch. **User defined exceptions in java** are also known as **Custom exceptions**.

All exceptions must be a child of Throwable.

If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.

If you want to write a runtime exception, you need to extend the RuntimeException class.

Program1:

```
class MyException extends Exception{
   String str1;
   MyException(String str2) {
     str1=str2;
   }
   public String toString(){
     return ("Output String = "+str1) ;
   }
}
class CustomException{
   public static void main(String args[]){
     try{
       throw new MyException("Custom");
       // I'm throwing user defined custom exception above
     }
     catch(MyException exp){
       System.out.println("Hi this is my catch block") ;
       System.out.println(exp) ;
     }
   }
}
```

Output:

```
Hi this is my catch block
Output String = Custom
```

<u>Points to Remember</u>

- Extend the Exception class to create your own exception class.
- You don't have to implement anything inside it, no methods are required.
- You can have a Constructor if you want.
- You can override the toString() function, to display customized message.

<u>Program2:</u>

```
class MyOwnExceptionClass extends Exception {
    private int price;
    public MyOwnExceptionClass(int price)
           {
            this.price = price;
            }
           public String toString()
           {
            return "Price should not be in negative, you are entered" +price;
            }
 }
public class Client {
public static void main(String[] args)throws Exception
{
  int price = -120;
  if(price < 0)
    throw new MyOwnExceptionClass(price);
  else
    System.out.println("Your price is :"+price);
  }
}
```

Program3:

```
class InvalidAgeException extends Exception{
 InvalidAgeException(String s){
  super(s);
 }
}
class TestCustomException1{
   static void validate(int age)throws InvalidAgeException{
   if(age<18)
    throw new InvalidAgeException("not valid");
   else
    System.out.println("welcome to vote");
  }
  public static void main(String args[]){
    try{
    validate(13);
    }catch(Exception m){System.out.println("Exception occured: "+m);}

    System.out.println("rest of the code...");
 }
}
```

Rules:
1. We can’t have catch or finally clause without a try statement.
2. A try statement should have either catch block or finally block, it can have both blocks.
3. We can’t write any code between try-catch-finally block.
4. We can have multiple catch blocks with a single try statement.
5. Try-catch blocks can be nested similar to if-else statements.
6. We can have only one finally block with a try-catch statement.
7. If you do not explicitly use the try catch blocks in your program, java will provide a default exception handler, which will print the exception details on the terminal, whenever exception occurs.
8. Super class **Throwable** overrides **toString()** function, to display error message in form of string.
9. While using multiple catch block, always make sure that exception subclasses comes before any of their super classes. Else you will get compile time error.

10. In nested try catch, the inner try block, uses its own catch block as well as catch block of the outer try, if required.
11. Only the object of Throwable class or its subclasses can be thrown.

:Summary:
An exception is an event, which occurs during the execution of a program, that interrupts the normal flow of the program. It is an error thrown by a class or method reporting an error in code.
The ***'Throwable'*** class is the superclass of all errors and exceptions in the Java language
Exceptions are broadly classified as ***'checked exceptions'*** and ***'unchecked exceptions'***. All RuntimeExceptions and Errors are unchecked exceptions. Rests of the exceptions are called checked exceptions. Checked exceptions should be handled in the code to avoid compile time errors.
Exceptions can be handled by using ***'try-catch'*** block. Try block contains the code which is under observation for exceptions. The catch block contains the remedy for the exception. If any exception occurs in the try block then the control jumps to catch block.
If a method doesn't handle the exception, then it is mandatory to specify the exception type in the method signature using ***'throws'*** clause.
We can explicitly throw an exception using ***'throw'*** clause

5.5 Difference between final, finally and finalize

No.	Final	Finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used toplace important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

1)Java final example
```
class FinalExample{
public static void main(String[] args){
final int x=100;
x=200;//Compile Time Error
}}
```

2)Java finally example
```
class FinallyExample{
public static void main(String[] args){
try{
int x=300;
}catch(Exception e){System.out.println(e);}
finally{System.out.println("finally block is executed");}
}}
```

3) Java finalize example

```
class FinalizeExample
{
protected void finalize()
{
System.out.println("finalize called");
}
public static void main(String[] args)
{
FinalizeExample f1=new FinalizeExample();
FinalizeExample f2=new FinalizeExample();
f1=null;
f2=null;
System.gc();
}
}
```

5.6 Basics about Strings in Java

1. You can create Strings in various ways:-

a) By Creating a String Object

```
String s=new String("abcdef");
```

b) By just creating object and then referring to string

```
String a=new String();
a="abcdef";
```

c) By simply creating a reference variable

```
String a="abcdef";
```

2.All the strings gets collected in a special memory are for Strings called " String constant pool".

3. JVM does all string related tasks to avoid the memory wastage for more info on this refer "How JVM Handles strings".

4. Every String is considered as a string literal.

5. Strings are immutable only reference changes string never changes. New string literals are referenced when there is any manipulation. The old string gets lost in the preceding. Look at example: String s="abcd";

```
s=s+"efgh";
```

5.7 String Methods and Operations

Following are most commonly used methods in the String class.

1. public String concat(String s)

This method returns a string with the value of string passed in to the method appended to the end of String which used to invoke the method.

```
String s="abcdefg";
System.out.println(s.concat("hijlk"));
```

Note:-Always use assignment operator in case of concat operator otherwise concat will be unreferenced and you will get old String.example

```
s.concat("hijkl");
System.out.println(s);
```

It will present output as " abcdefg " different than what we have expected. So always be careful in using the assignment operator in String method calls.

2. Character Extraction methods:

A. char charAt(int index)

This method returns a specific character located at the String's specific index. Remember,String indexes are zero based.

Example

```
String s="Alfanso Mango";
System.out.println(s.charAt(0));
```

The output is 'A'

B. void getchars(int sourcestart, int sourceend,char trgt[],int trgtstart)

If you need to extract more than one character at a time than you can use getchars()method.

Example:

```
class gdemo{
public static void main(String a[])
   {
        String s = “this is demo program”;
        int start = 10;
        int end = 14;
        char str[] = new char[end – start];

        s.getChars(start,end,str,0);
        System.out.println(str);
   }
}
```

The output is mo p

3. public int length()
This method returns the length of the String used to invoke the method.
Example:-
```
String s="name";
System.out.println(s.length());
```

The output is 4

4. public String replace(char old,char new)
This method return a String whose value is that of the String to invoke the method ,updated so that any occurrence of the char in the first argument is replaced by the char in the second argument .
Example:-
```
String s="VaVavavav";
System.out.println(s.replace('v','V'));
```

The output is VaVaVaVaV

5. String Comparison Methods:

A. public boolean equals(String s/object str) & public boolean equalsIgnoreCase (String s/object str)

equals method returns true if the string contain the same characters in the same order, and false otherwise. Here the comparison is case-sensitive. Example:

```
String s="Vaibhav";
System.out.println(s.equals("VAIBHAV"));
```

The output is false

equalsIgnoreCase method returns a boolean value depending on whether the value of the string in the argument is the same as the String used to invoke the method. This method will return true even when character in the string object being compared have different cases.Example:-

```
String s="Vaibhav";
System.out.println(s.equalsIgnoreCase("VAIBHAV"));
```

The output is true

B. regionMatches()

This method compares specific region inside a string with another specific region in another string.
Syntax:
boolean regionMatches(int startindex, String str, int str2startindex, int numChar)
OR
boolean regionMatches(boolean ignoreCase, int startindex, String str, int str2startindex, int numChar)

in second case if ignoreCase is true., the case of the character is ignored. If it is false then vise versa. Example:

```
String s1="Vaibhav";
String s2="Naibhav";
System.out.println(s1.regionMatches(1,s2,1,6));
o/p: true
```

```
String s1="Vaibhav";
String s2="NAibhav";
System.out.println(s1.regionMatches(false,1,s2,1,6));
o/p: false
```

C. startsWith() & endsWith():

boolean startsWith(String str)
Example:
```
String s = "Arth Kavi and Pihu";
System.out.println(s.startsWith("Arth"));
o/p: true
```

boolean startsWith(String str, int statrtIndex)
Example:
```
String s = "Arth Kavi and Pihu";
System.out.println(s.startsWith("Arth",4));
o/p: false
```

Both the forms of startsWith methods return true if given string begins with a specified string. The second form of startsWith method lets you specify a starting point.

boolean endsWith(String str)
endsWith method return true if given string ends with a specified string.
Example:
```
String s = "Arth Kavi and Pihu";
System.out.println(s.endsWith("Arth"));
```

```
o/p: false

String s = "Arth Kavi and Pihu";
System.out.println(s.endsWith("Pihu"));
o/p: true
```

D. int compareTo(String str):

In this method given string str is being compared with the invoking string.
It will return <0 value if the invoking string is less than str.
It will return >0 value if the invoking string is greater than str.
It will return 0 value if both strings are equal.
Ex:

```
String s1 = "Arth";
String s2 = "Kavi";
System.out.println(s1. compareTo (s2));
o/p: -10
String s1 = "Arth";
String s2 = "Aavi";
System.out.println(s1. compareTo (s2));
o/p: 17
String s1 = "Kavi";
String s2 = "Kavi";
System.out.println(s1. compareTo (s2));
o/p: 0
```

Note: equals() versus = =

Equals() method compares character inside string object whereas = = operator compares 2 object references to see whether they refer to same instance. Example:

```
String s1 = "Kavi";
String s2 = new String(s1);
System.out.println("equals:"+s1. equals (s2));
System.out.println("==:"+(s1==s2));
o/p: equals: true
     = = : false
```

6. public String substring(int begin) / public String substring(int begin,int end)

substring method is used to return a part or substring of the String used to invoke the method. The first argument represents the starting location of the substring. Remember the indexes are zero based. example:-

```
String s="abcdefghi";
System.out.println(s.substring(5));
System.out.println(s.substring(5,8));
```

The output would be
" fghi "
" fgh"

7. public String toLowerCase()
This method returns a string whose value is the String used to invoke the method, but with any uppercase converted to lowercase.:-

```
String s="AbcdefghiJ";
System.out.println(s.toLowerCase());
```

Output is " abcdefghij "

8. public String trim()
This method returns a String whose value is the String used to invoke the method ,but with any leading or trailing blank spaces removed. Example:-

```
String s="hey here is the blank space ";
System.out.println(s.trim())
```

The output is " hey here is the blank space"

9. public String toUpperCase()
This method returns a String whose value is String used to invoke the method ,but with any lowercase character converted to uppercase.Example:-

```
String s="AAAAbbbbb";
System.out.println(s.toUpperCase());
```

The output is " AAAABBBB"

Above mentioned String methods are most commonly used in String class. Do remember them or otherwise I am always up for you and of course you can refer this site every time you need something.
10. Search String:
A. int indexOf(String str) & int indexOf(String str, int stratindex)
It search for the first occurrence of a character or substring(piece of string). Example;

```
String s = "Arth Kavi and Pihu";
System.out.println(s.indexOf("a"));
o/p: 6
```

It search for the last occurrence of a character or substring(piece of string). Example;

```
String s = "Arth Kavi and Pihu";
System.out.println(s.lastIndexOf("a"));
o/p: 10
```

5.8 String Buffer Class

StringBuffer is the class included in java.lang and is mostly used when we have to make a lot of modifications to Strings of characters. As (only) String class does not support string mutability(changeability) . In short when we use Java Strings class our strings becomes Immutable(absolute) and can never be changed.

[If we use String class we are left with numerous dangling (hanging) strings in the memory which not only waste the memory but could also cause the memory overflow at some stage.]

Objects of StringBuffer class can be modified a number of times and even you don't need a reference to the object to assign. Simply use the method on the object and change will reflect on the object

The main use of StringBuffer class is in File I/O where large stream of data constantly changing values. Here the loss of memory never happens. Large block of characters are handled as units ,and StringBuffer objects are the ideal way to handle a block of data, pass it on and then reuse the memory block to handle the next stream of data.

Constructors:-

1. StringBuffer() //**Creates a StringBuffer with initial capacity of 16 characters.**

2. StringBuffer(String s) //Creates StringBuffer which contains same no of characters as in the String argument.

3. StringBuffer(int length) //Creates StringBuffer with specified number of characters.

Lets see an example on StringBuffer

```
// don't forget to import the StringBuffer class(java.lang.StringBuffer.*;)
StringBuffer x = new StringBuffer("ABCDE");
x.append("FGH");
System.out.println(x);
```

Output would be :- ABCDEFGH

The main feature of StringBuffer methods are as follows :-

The main feature of this StringBuffer class is that all methods are thread safe that is marked as synchronized. With synchronized marking StringBuffer comes with a disadvantage that it works slower. To overcome the problem of this Thread safe and slow execution we use StringBuilder class.

In short, The StringBuffer class is used to represent characters that can be modified. This is simply used for concatenation or manipulation of the strings.

[*StringBuffer* is mainly used for the dynamic string concatenation which enhances the performance. A string buffer implements a mutable sequence of characters. A string buffer is like a String, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.]

There are some functions used in the given example. All the functions have been explained below with example:

append()
This is the append() function used for the concatenate the string in string buffer. This is better to use for dynamic string concatenation. This function works like a simple string concatenation such as : String str = str + "added string";.
Syntax: StringBuffer append(String str)

insert()
This is the insert() function used to insert any string or character at the specified position in the given string. Syntax: StringBuffer insert(int index,String str)

reverse()
This is the reverse() function used to reverse the string present in string buffer.
Syntax: StringBuffer reverse()

setCharAt()
This is the setCharAt() function which is used to set the specified character in buffered string at the specified position of the string in which you have to set the given character.
Syntax: StringBuffer setCharAt(int where, char ch)

charAt()
This is the charAt() function which is used to get the character at the specified position of the given string. Syntax: StringBuffer charAt(int where)

substring()
This is the substring() function which is used to get the sub string from the buffered string from the initial position to end position (these are fixed by you in the program).
Syntax: String substring(int begin,int end) and String substring(int begin)

deleteCharAt()

This is the deleteCharAt() function which is used to delete the specific character from the buffered string by mentioning that's position in the string.
Syntax: StringBuffer deleteCharAt(int loc)

length()
This is the length() function is used to finding the length of the buffered string.
Syntax: int length()

delete()
This is the delete() function is used to delete multiple character at once from *n* position to *m* position (n and m are will be fixed by you.) in the buffered string.
Syntax: StringBuffer delete(int start, int end)

capacity()
This is the capacity() function is used to know about the current characters kept which is displayed like : number of characters + 16. Syntax: int capacity()

In following example you will learn about StringBuffer class. This example explains how you can use functions provided by the StringBuffer class like *append, insert, reverse, setCharAt, charAt, length, deleteCharAt, substring, delete, capacity* etc. to manipulate the string operation in your program.

<u>Program</u>:

```
import java.io.*;
public class stringBuffer{
 public static void main(String[] args) throws Exception{
  BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
  String str;
  try{
   System.out.print("Enter your name: ");
   str = in.readLine();
        StringBuffer strbuf1 = new StringBuffer(str);
        str += ", This is an example of StrngBuffer Class.";

        //Create a object of StringBuffer class
   StringBuffer strbuf = new StringBuffer();
   strbuf.append(str);
   System.out.println(strbuf);
   strbuf.delete(0,str.length());

   //append()
   strbuf.append("Hello");
   strbuf.append("World");           //print HelloWorld
```

```
    System.out.println(strbuf);

    //insert()
    strbuf.insert(5,"_Java ");          //print Hello_Java World
    System.out.println(strbuf);

    //reverse()
    strbuf.reverse();
    System.out.print("Reversed string : ");
    System.out.println(strbuf);         //print dlroW avaJ_olleH
    strbuf.reverse();
    System.out.println(strbuf);         //print Hello_Java World

    //setCharAt()
    strbuf.setCharAt(5,' ');
    System.out.println(strbuf);         //prit Hello Java World

    //charAt()
    System.out.print("Character at 6th position : ");
    System.out.println(strbuf.charAt(6));     //print J

    //substring()
    System.out.print("Substring from position 3 to 6 : ");
    System.out.println(strbuf.substring(3,7));   //print lo J

    //deleteCharAt()
    strbuf.deleteCharAt(3);
    System.out.println(strbuf);         //print Helo java World

    //capacity()
    System.out.print("Capacity of StringBuffer object : ");
    System.out.println(strbuf1.capacity());     //print 21

    //delete() and length()
    strbuf.delete(6,strbuf.length());
    System.out.println(strbuf);         //Helo J
   }
   catch(StringIndexOutOfBoundsException e){
    System.out.println(e.getMessage());
   }
  }
}
```

output:
```
Enter your name: kejal
kejal, This is an example of StringBuffer Class.
HelloWorld
Hello_Java World
Reversed string : dlroW avaJ_olleH
Hello_Java World
Hello Java World
Character at 6th position : J
Substring from position 3 to 6 : lo J
Helo Java World
Capacity of StringBuffer object : 21 (5 for kejal + 16 default capacity value)
Helo J
```

<u>5.9</u> <u>Exercise:</u>

1. Explain user defined exception with an example.
2. Explain exception types in detail.
2. Describe exception handlers in java.
3. Explain constructor of String class along with any 5 methods with examples.
4. Explain StingBuffer class along with any 5 methods with examples.
5. Differentiate followings:
    1. length and capacity
    2. equals and ==
    3. String and StringBuffer

6. Threads and the Applet

6.1 Introduction to Thread:

Program in the state of execution is called as a process and the operating system which allows more than one process to execute at a same time is called as multi processing. **A piece or set of code in the program is called thread and the program which allows such blocks of statement to execute at the same time is called multi threading.** In short a "multithreaded program contains two or more parts that can run concurrently. Multithreading is a specialized form of multitasking".

Take an example of windows environment, which allows notepad, word, excel, etc to open up at the same time and allows execution of these program at the same time is called multi processing and the CPU which handle this is called heavy weight process execution.

Now take an example of video game, where process has to take care of movement of object, audio signals and increment of scores. Although these 3 things come in a single program, each task should be executed in its own environment and the user should feel that these entire tasks occur at the same time. This is an example of multi threading. **It is a light weight process**. Multithreading enables you to write efficient program and make maximum use of CPU time because ideal time can be kept minimum.

6.2 Creating Thread:

You can create thread in java using 2 ways:

1. By creating a class which **extends Thread** Class
2. By creating a class which **implements Runnable** Interface.

Choice of using either extends of Thread class or implementing Runnable interface depends upon requirement. if we require to extend another class then we have no choice but to implement Runnable interface because java can't have 2 super classes.

Thread class & Runnable interface overrides run() method. You can write necessary code required by the thread in this method.

```
Public void run()
{
// statement to implementing thread
}
```

Run() method can call other methods, use other classes & declare variable just like main thread.

Example:

Let's take an example of a simple program which prints time in following manner

```
hh:mm:ss
0:0:0
0:0:1
:
:
0:0:59
0:1:0
```

```
public class demo extends Thread
{
public static void main(String args[])
{
int hh,mm,ss;
try
{
        Thread t=new Thread("demo");
        for(hh=0;hh<24;hh++)
                {
                for(mm=0;mm<=59;mm++)
                        {
                        for(ss=0;ss<=59;ss++)
                                {
                                System.out.println(hh+":"+mm+":"+ss);
                                t.sleep(1000);
                                }//ss
                        }//mm
                }//hh
}//try
catch(Exception e)
{
System.out.println("Error:" + e);
}//catch

}//main
}//class
```

6.2.1 Implements Runnable:

Runnable interface has only one method run(). Once you declared any class which implements Runnable interface, you have to create object of thread type. There are several constructors to define thread but one of them is:

Thread(Runnable threadobject, String threadname)

Ex: Thread t = new Thread(this,”demo thread”)

Example of implementing Runnable interface:

```
import java.lang.Thread;
class demo implements Runnable
{
        demo()
        {
        Thread t = new Thread(this,"demo thread");
        System.out.println("Child thread"+t);
        t.start();
        }
        public void run()
        {
                try{
                        for (int i =5;i>0;i--)
                        {
                        System.out.println("child thread:"+i);
                        Thread.sleep(500);
                        }
                  }//try

                catch(InterruptedException e)
                  {
                        System.out.println("Child Interrupted");
                  }//catch
        System.out.println("Exit from child");

        }
};
class drdemo
{
public static void main(String a[])
{
demo d = new demo();// call to constructor.
//or
//new demo();// call to constructor.
try
        {
                for (int i =5;i>0;i--)
                        {
                        System.out.println("Main thread:"+i);
                        Thread.sleep(1000);
                        }
        }//try


        catch(InterruptedException e)
        {
                        System.out.println("Main Interrupted");
        }//catch
        System.out.println("Exit from main");
}
};
```

We can not use super("demo thread") because Runnable is an interface & not a class.
We can not use only start() method because Runnable interface has one & only one method i.e. run().

6.2.2 Extending Thread:

To extends thread;

1. Create class extending Thread class
2. Implement run() method
3. Create an object & call the start method to initiate thread execution.

To create and run instance of thread class use following statments:

Demo d1 = new Demo(); $\longrightarrow$[Equivalent to **new Demo();**]
d1.start();
OR
new Demo() . start(); (compact nature of above two statement)

Example of extending Thread class:

```
import java.lang.Thread;
```

```
class demo extends Thread
{
        demo()
        {
        super("demo thread");           //or Thread t = new Thread(this,"demo thread");
        System.out.println("Child thread"+this);// or System.out.println("Child thread"+t);
        start(); // or t.start();
        }

        public void run()
        {
                try{
                        for (int i =5;i>0;i--)
                        {
                        System.out.println("child thread:"+i);
                        Thread.sleep(500);
                        }
                  }//try

                catch(InterruptedException e)
                  {
                        System.out.println("Child Interrupted");
                  }//catch
        System.out.println("Exit from child");

        }
};
class dtdemo
{
public static void main(String a[])
{
demo d = new demo();// call to constructor.
//      OR
// new demo();// call to constructor.
try
        {
                for (int i =5;i>0;i--)
                        {
                        System.out.println("Main thread:"+i);
                        Thread.sleep(1000);
                        }
        }//try

        catch(InterruptedException e)
        {
```

```
                    System.out.println("Main Interrupted");
```

```
        }//catch
        System.out.println("Exit from main");
}
};
```

1. Because Thread is a class, we can use "super" keyword to access Thread class's property. here super("any thread name") will call Thread class constructor & allocates specific name to current thread.   Ex. super("demo thread") allocates "demo thread" name to current thread & which is similar to Thread t = new Thread(this,"demo thread");
2. To access current thread we can make use of "this" keyword Ex. System.out.println("Child thread"+this);
3. Once we have directly access thread without any object, we can directly call start methd Ex. start();

6.2.3 Multi threading:
An example which demonstrates, more than 2 threads running at the same time.

Example:

```
import java.lang.Thread;
class demo implements Runnable
{
String name;
Thread t;
        demo(String tname)
        {
        name=tname;
        t = new Thread(this,name);
        System.out.println("New Child thread"+t);
        t.start();
        }
        public void run()
        {
                try{
                        for (int i =5;i>0;i--)
                        {
                        System.out.println(name+":"+i);
                        Thread.sleep(1000);
                        }
                  }//try

                catch(InterruptedException e)
                  {
                        System.out.println(name+"Interrupted");
                  }//catch
        System.out.println(name+" Exiting");
        }
};

class mrdemo
{
public static void main(String a[])
{
new demo("one"); // demo d1 = new demo("one");
new demo("two"); // demo d2 = new demo("two");
new demo("three"); // demo d3 = new demo("three");

        try
        {
                Thread.sleep(10000);
        }//try


        catch(InterruptedException e)
        {
                        System.out.println("Main Interrupted");
        }//catch

        System.out.println("Main Exiting");
}
};
```

6.3 Methods

The Thread class defines several methods that help manage threads. The table below displays the same:

Method	Meaning
getName	It returns the name of the thread.
getPriority	It returns the priority of the thread.
isAlive	It determines if a thread is still running
Join	Wait for a thread to terminate
Run	Entry point for the thread
Sleep	Suspend a thread for a period of time
Start	Start a thread by calling its run method
setPriority	It changes the priority of the thread.
yield()	It causes current thread on halt and other threads to execute.
setName	It changes the name of the thread

Note: Rather than above methods, there is one more thread which work as a service provider thread name as Daemon thread.

Daemon thread: daemon thread is a service provider thread that provides services to the user thread. It is a low priority thread. Its life depend on the user threads i.e. when all the user threads dies, JVM terminates this thread automatically. It has no role in life than to serve user threads. It provides services to user threads for background supporting tasks. There are many java daemon threads running automatically e.g. gc, finalizer etc.

Daemon thread methods by Thread class

The java.lang.Thread class provides two methods for java daemon thread.

No.	Method	Description
1)	public void setDaemon(boolean status)	It is used to mark the current thread as daemon thread or user thread.
2)	public boolean isDaemon()	It is used to check that current is daemon.

getName & setName example:

```
import java.lang.Thread;
class demo implements Runnable
{
String name;
Thread t;
        demo(String tname)
        {
        name=tname;
        t = new Thread(this,name);
        System.out.println("New Child thread:"+t.getName());//one
        t.setName("Thread:"+name);
        System.out.println("New Name of thread is:"+t.getName());//Thread:one
        t.start();
        }
        public void run()
        {
                try{
                        for (int i =5;i>0;i--)
                        {
                        System.out.println(t.getName()+":"+i);
                        Thread.sleep(1000);
                        }
                  }//try

                catch(InterruptedException e)
                  {
                        System.out.println(t.getName()+"Interrupted");
                  }//catch
        System.out.println(name+" Exiting");

        }
};
class mrgetset
{
public static void main(String a[])
{
new demo("one"); // demo d1 = new demo("one");
new demo("two"); // demo d2 = new demo("two");
new demo("three"); // demo d3 = new demo("three");

try
        {
                Thread.sleep(10000);
        }//try


        catch(InterruptedException e)
        {
                        System.out.println("Main Interrupted");
        }//catch

        System.out.println("Main Exiting");
}
};
```

Using isAlive() & join():

We generally used sleep() within main to finish the main thread at last. This is used to ensure that all child thread s terminate prior to the main thread. However it will hardly work in predictable way & it will raise a question: how one thread know when another thread has ended?
However java (thread) provides solution for this question.
There is 2 way to determine whether thread has finished.
First is isAlive() & second is join() method.

isAlive():

This method return true if the thread upon which it is called is still running, otherwise it will return false.
General form of this method is:
Final boolean isAlive()

Join():

This method waits until thread on which it is called terminates. Its name comes from the concept of calling thread waiting until the specified thread joins it.

Additional form of join() allows you to specify maximum amount of time on specified thread to terminate. General form of this method is:
Final void join() throws InterruptedException

join Example:

```
import java.lang.Thread;
class callme
{
        void call(String msg)
        {
        System.out.println("["+msg);
                try
                {
                        Thread.sleep(1000);
                }
                catch(InterruptedException e)
                {
                        System.out.println("Interrupted:");
                }
        System.out.println("]");
        }
}
class demo implements Runnable
{
String  msg;
callme trgt;
Thread t;
public demo(callme targ,String s)
        {
        trgt=targ;
        msg=s;
        t=new Thread(this);
        t.start();
        }
public void run()
        {
                trgt.call(msg);
        }
}// class demo

class join
{
public static void main(String a[])
{
```

```
callme target = new callme();
demo d1 = new demo(target,"Hello");
demo d2 = new demo(target,"Synchronized");
demo d3 = new demo(target,"World");
try
{
        d1.t.join();
        d2.t.join();
        d3.t.join();
}
catch(InterruptedException e)
{
        System.out.println("Interrupted");
}//catch
}//main
};
```

Here is the o/p produce by this program:
[hello[synchronized[world]]] // it may be different.

getName/setName/isAlive/join example:

```
import java.lang.Thread;
class demo implements Runnable
{
String name;
Thread t;
        demo(String tname)
        {
        name=tname;
        t = new Thread(this,name);
        System.out.println("New Child thread:"+t.getName());
        t.setName("Thread:"+name);
        System.out.println("New Name of thread is:"+t.getName());
        t.start();
        }

        public void run()
        {
                try{
                        for (int i =5;i>0;i--)
                        {
                        System.out.println(t.getName()+":"+i);
                        Thread.sleep(1000);
                        }
```

```
                  }//try
                catch(InterruptedException e)
                  {
                        System.out.println(t.getName()+"Interrupted");
                  }//catch
        System.out.println(name+" Exiting");

        }
};
class mrgsaj
{
public static void main(String a[])
{
 demo d1 = new demo("one");
 demo d2 = new demo("two");
 demo d3 = new demo("three");

System.out.println(d1.t.isAlive());
System.out.println(d2.t.isAlive());
System.out.println(d3.t.isAlive());

        try
        {
                d1.t.join();
                d2.t.join();
                d3.t.join();
                Thread.sleep(10000);
        }//try


        catch(InterruptedException e)
        {
                        System.out.println("Main Interrupted");
        }//catch

        System.out.println("Main Exiting");
}
};
```

<u>6.4</u> <u>Race condition:</u>

Race condition in **Java** occurs in a multi-threaded environment when more than one thread try to access a shared resource (modify, write) at the same time.

Let’s say, When more than one thread want to complete it task in that process then it simply snatches the control of the CPU so intern you can’t specify which process will execute first?, because all threads are in hurry to complete its process which is called as race condition.

6.5 Synchronization:

When 2 or more threads need to access a shared resources they need some way to ensure that the resources will be used by **only 1 thread at a time**. The process by which it is achieved is called synchronization.

Key to synchronization is the concept of the monitor (also called semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All the other threads attempting to enter a locked monitor will be suspended until the first thread exists the monitor. These other threads are said to be waiting for the monitor, so here instead of getting the garbage output on monitor. By giving time slice to each thread, java helpsto implement the synchronization (which allows to have mutex). A thread that owns a monitor can reenter the same monitor if it so desire.

You can synchronize your code in either of 2 ways. Both involve the use of the synchronized keyword.

1. By using synchronized method
2. By using synchronized statement

By using synchronized method:

Earlier example produce o/p like: [hello[synchronized[world]]]. This can be achieve by calling sleep(), the call() method allows execution to switch to another thread. This result in mixed o/p. to fix the o/p of preceding program you must serialize access to call(). That is, you must restrict its access to only 1 thread at a time. To do so, you simply need to precede call()'s definition with the keyword synchronized.

```
Ex:   class callme {
                synchronized void call(String msg){ .............}
      }
```

Example:

```
import java.lang.Thread;
class callme
{
        synchronized void call(String msg)
        {
        System.out.println("["+msg);
                try
                {
                        Thread.sleep(1000);
                }
                catch(InterruptedException e)
                {
                        System.out.println("Interrupted:");
                }
        System.out.println("]");
        }
}

class demo implements Runnable
{
String  msg;
callme trgt;
Thread t;

public demo(callme targ,String s)
        {
        trgt=targ;
        msg=s;
        t=new Thread(this);
        t.start();
        }
public void run()
        {
                trgt.call(msg);
        }
}// class demo

class synchro
{
public static void main(String a[])
{
callme target = new callme();
demo d1 = new demo(target,"Hello");
demo d2 = new demo(target,"Synchronized");
demo d3 = new demo(target,"World");
try
{
        d1.t.join();
        d2.t.join();
        d3.t.join();
}
catch(InterruptedException e)
{
        System.out.println("Interrupted");
}//catch
}//psvm
};
```

After synchronized has been added to call(), the o/p of the program is as follows:

```
[hello][synchronized][world]
```

By using synchronized statement:

Using synchronized keyword to methods will not work in all the cases. Consider the case, that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use synchronized methods. Further this class was not created by you, but by the third party & you don't know how to access the source code. Thus you can't add synchronized to the methods of that class. How to synchronize access to an object of this class? But java provides solution to this problem: you put calls to the methods defined by this class inside synchronized block.

Syntax: synchronized(object){ ……. }

```
Ex: public void run()
        {
            synchronized(trgt){
                                        trgt.call(msg);
                                    }
        }
```

Example:

```
import java.lang.Thread;
class callme
{
        void call(String msg)
        {
        System.out.println("["+msg);
                try
                {
                        Thread.sleep(1000);
                }
                catch(InterruptedException e)
                {
                        System.out.println("Interrupted:");
                }
        System.out.println("]");
        }

}
class demo implements Runnable
{
String  msg;
callme trgt;
Thread t;
public demo(callme targ,String s)
        {
```

```
        trgt=targ;
        msg=s;
        t=new Thread(this);
        t.start();
        }
public void run()
        {
            synchronized(trgt){
                                    trgt.call(msg);
                                }
        }

}// class demo

class synchro
{
public static void main(String a[])
{
callme target = new callme();

demo d1 = new demo(target,"Hello");
demo d2 = new demo(target,"Synchronized");
demo d3 = new demo(target,"World");

try
{
        d1.t.join();
        d2.t.join();
        d3.t.join();
}

catch(InterruptedException e)
{
        System.out.println("Interrupted");
}//catch
}//psvm
};
```

This program will produce the same o/p like:

```
[hello][synchronized][world]
```

6.6 Thread priority:

Thread priority are used by scheduler to decide when each thread should be allowed to run. In theory, higher priority thread gets more CPU time than low priority thread but in practice the amount of CPU time gets by any thread depends upon several factors.
The higher priority thread can also preempt lower priority thread, when lower priority thread is running & higher priority thread resumes, it will preempt lower priority thread.
In theory, threads of equal priority get equal access to the CPU. But you need to be careful because java designed to works in wide environment. Threads that shares same priority should yield (surrender/give up) control once in while. And this situation may leads to race condition & your program may behave differently because of this kind of situation. You can control behavior through setting priority to threads.

To set thread's priority, use the **setPriority()** method, which is a member of thread. General form of this method is:
final void setPriority(int level)
Here level specifies the new priority for the calling thread. The value for the level must be within the range MIN_PRIORITY and MAX_PRIORITY. Currently these values are 1and 10 respectively. To return thread to default priority use NORM_PRIORITY, which is currently 5.You can obtained the current priority setting by calling the **getPriority()** method of Thread. General form of this method is:
final int getPriority()

6.7 Interthread communication:

Multithreading replaces event loop programming by dividing your task into discrete & logical unit. Threads also provide secondary benefit to do away with polling. Polling is usually implemented by loop. i.e. Used to check some condition repeatedly. Once the condition is true appropriate action is taken. This wastes CPU time. This situation is undesirable.
[For example…… Let us consider producer & consumer problem. Now suppose that producer has to wait for generating data until consumer finishes his job. In polling system, consumer wasting many CPU cycle while it waited for the producer to produce. & once producer was finished, it would start polling **(asking/questioning)**, wasting more CPU cycle waiting for consumer to finish & so…on.]

To avoid polling, java includes an interprocess communication mechanism through **wait(),notify(),notifyAll().** These methods are the final methods in object (so all class have them) & should be call only within the synchronized method.

1. wait(): tells the calling thread to give up the monitor & go to sleep until some other thread enters the same monitor & calls notify().
   final void wait() throws Interrupted Exception
2. notify(): wakes up the first thread that called wait() on the same object.
   final void notify()
3. notify All(): wakes up all the threads that called wait() on the same object.
   final void notifyAll()

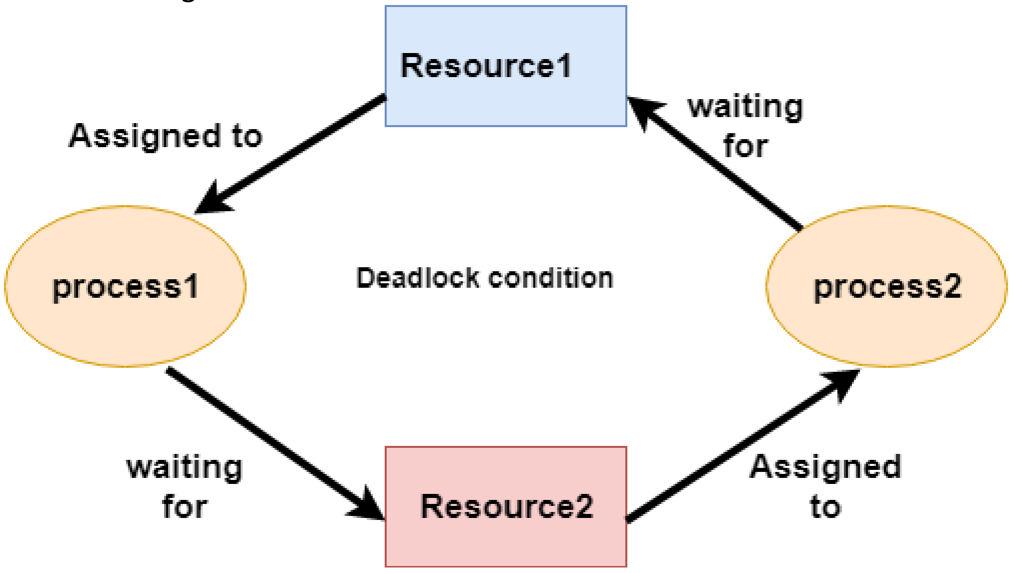
Additional forms of wait() exist which allows you to specify a period of time to wait.

6.8 Deadlock:

Deadlock in java is a part of multithreading, which occurs when two threads have a circular dependency on a pair of synchronized objects.

Deadlock in Java is a situation that occurs when a thread is waiting for an object lock that is acquired by another thread and second thread is waiting for an object lock that is acquired by the first thread.

Since both threads are waiting for each other to release the lock simultaneously, this condition is called deadlock in Java. The object locks acquired by both threads are not released until their execution is not completed. For example Process1 is holding Resource1 and waiting for resource2, which is acquired by process2, and process2 is waiting for resource1 is explained in the below diagram:



In above diagram process1 and process2 are two threads that are in a deadlock. The process1 holds the lock for the resource R1 and waits for resource R2 that is acquired by the process2. At the same time, the process holds the lock for the resource R2 and waits for R1 resource that is acquired by the process1. But process2 cannot release the lock for resource R2 until it gets hold of resource R1.

Since both threads are waiting for each other to unlock resources R1 and R2, therefore, these mutually exclusive conditions are called deadlock in Java.

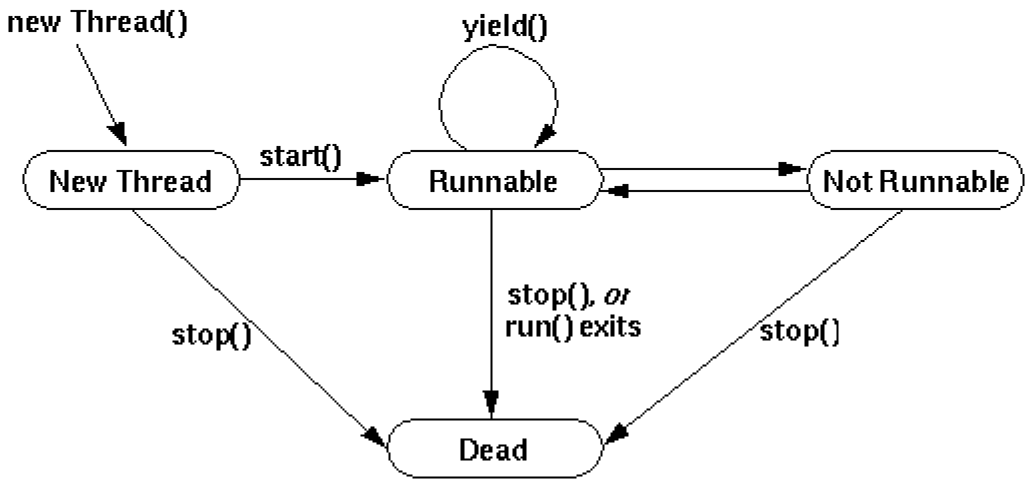
How to avoid/release deadlock:

1. A deadlock in a program can be prevented if any of the above four conditions are not met.
2. It can be achieved if a thread is to force to hold only one resource at a time. If it needs another resource, it must first release that resource that is held by it and then requests another.
3. It can be achieved by acquiring resources (locks) in a specific order and releasing them in reverse order so that a thread can only continue to acquire a resource if it held the other one.

Deadlock is a difficult error to debug for two reasons:

1. In general, it occurs only rarely, when the two threads time-slice in just the right way.
2. It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of events than just described.)

6.9 Thread State:



New Thread:
The following statement creates a new thread but does not start it, thereby leaving the thread in the "New Thread" state.
Thread t = new Thread();
It is an empty thread. No system resources have been allocated for it yet. So you can only start or stop this thread.

Runnable:
Consider the statement t.start()in following 2 lines of code. It will start the thread and put thread in runnable state.
Thread t = new Thread();
t.start();
The start() method allocates system resources, schedules the thread, and calls the threads's run() method and not actually run the method. When the thread actually *runs* is determined by the **scheduler**

Not Runnable:
A thread enters the "Not Runnable" state when one of these four events occurs:

- Someone invokes its sleep() method.
- Someone invokes its suspend() method.
- The thread uses its wait() method to wait on a condition variable.
- The thread is blocking on I/O.

Dead:

- a thread enters in the **dead** state when it's run() method completes.
- An **interrupt** does not kill a thread.
- The **destroy ()** method kills a thread dead but does not release any of it's object locks.
- The stop () method causes a sudden termination of a Thread's run() method. For ex: t.stop();

6.10 Applet

Applets are small applications that are accessed on an internet server, transported over the internet, automatically installed & run as part of a web document. An applet is typically embedded inside a web-page and runs in the context of the browser. The applet class provides a standard interface between applets & their environment. An applet is nothing but a subclass of java.applet.Applet. Swing provides a special subclass of Applet, called javax.swing.JApplet, which should be used for all applets that use Swing components to construct their GUIs.

Ex:

```
/* <applet code="sapplet" width=200 height=60></applet> */
import java.awt.*;
import java.applet.*;
public class sapplet extends Applet
{
        public void paint(Graphics g)
        {
                g. drawString("hello",20,20);
        }
}
```

The applet begins with import system. The first statement imports abstract window toolkit (awt) classes. Where the awt contains support for a window based graphical interface. The second import statement imports the applet package, which contains the class Applet. Every applet that you create must be subclass of Applet.

The next line declarethe “sapplet” class where this class must be declared as public, because it will be accessed by code that is outside the program.

Inside “sapplet”, paint() method is declared.this method is defined by awt & overridden by applet. This method has one parameter type graphics, which describe graphic environment, in which applet is running.

Inside paint(), drawString function is call, this method outputs a string beginning at the specified x,y location.

void drawString(String msg, int x, int y)

Applet doesn’t have main method. Its execution begins when name of its class is passed to an applet viewer.

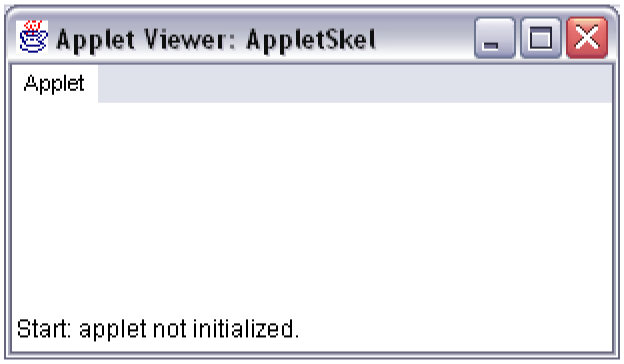
To execute an applet in web browser, you need to write a short html text that contains APPLET tag.

<applet code="sapplet" width=200 height=60></applet>
Where code specifies the name of the file which related to the base URL of the applet code. It is a .class compiled file. Width & height specifies dimension of display area used by an applet.
The applet HTML tag embedded in the web page using <applet></applet> tag. "sapplet" compile in the same way that you have been compiling other programs.

Compile: C:\> javac sapplet.java
Execute: C:\>appletviewer sapplet.java

Example of using javax.swing.

```
// An Applet skeleton.
import java.awt.*;
import javax.swing.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/

public class AppletSkel extends JApplet {
  // Called first.
  public void init() {
    // initialization
  }

  /* Called second, after init(). Also called whenever
     the applet is restarted. */
  public void start() {
    // start or resume execution
  }

  // Called when the applet is stopped.
  public void stop() {
    // suspends execution
  }

  /* Called when applet is terminated. This is the last
     method executed. */
  public void destroy() {
    // perform shutdown activities
  }

  // Called when an applet's window must be restored.
  public void paint(Graphics g) {
    // redisplay contents of window
  }
}
```



Start: applet not initialized.

6.11 Architecture of Applet:

Applet is window based program. As such its architecture is different from normal console based programs.

Applets are event driven program. It waits until an event occurs. The AWT notifies the applet about an event by calling event handler provided by the applet. Once this happens the applet must take appropriate action & then quickly return control to the AWT(runtime system). In those situation where applet need to perform some repetitive task on its own (ex. Displaying scrolling message across its windows) you must start an additional thread of execution.

The user initiates interaction with an applet & not the other way around. In non-window program. When program needs input it will prompt user & then call some input method such as readLine(). But applet takes different approach, Applet handle this with the help of some event/action to which applet must respond. Applet has various event llike mouse-clicked event, key-press event….etc Applet can contain various controls such as push button & check boxes, when user interacts with, one of these control, an event is generated.

Java AWT makes any window based program to response quickly.

6.12 Applet Initialization and Termination(skeleton / Life Cycle):

It is important to understand the order in which the various methods shown in the skeleton are called.

When an applet begins, the AWT calls the following methods, in this sequence:

1. init( ) //defined by applet component class
2. start( ) //defined by applet component class
3. paint( ) //defined by awt component class

When an applet is terminated, the following sequence of method calls takes place:

4. stop( ) //defined by applet component class
5. destroy( ) //defined by applet component class

Let’s look more closely at these methods.
init( )
The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

start( )
The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Whereas **init( )** is called once—the first time an applet is loaded—**start( )** is called each time an applet’s HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

paint( )
The **paint( )** method is called each time your applet’s output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called. The **paint( )** method has one parameter of type **Graphics**. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

stop( )
The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop( )** is called, the applet isprobably running. You should use **stop( )** to suspend threads that don’t need to run when the applet is not visible. You can restart them when **start( )** is called if the user returns to the page.

destroy( )
The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using (e.g. to kill any threads created in the init ()). The **stop( )** method is always called before **destroy()**.
Example:
Let now write an applet which really does something. Just something, not something useful!

```
/* This Applet sets the foreground and background colors and out puts a string. */
import java.awt.*;
import java.applet.*;
//Also watch command line for messages
/*<applet code="sapplet" width=600 height=50>
  </applet>
*/
public class sapplet extends Applet {
  String msg;
```

```
 // set the foreground and background colors.
 public void init()
          {
   setBackground(Color.cyan);
   setForeground(Color.red);
   msg = "Initialised --";
          }

   // Add to the string to be displayed.
 public void start()
          {
           msg += " Starting --";
           }

   // Display the msg in the applet window.
 public void paint(Graphics g)
           { msg += " Painting.";
           g.drawString(msg, 10, 30);
            }
      }
```

When you compile and run it output of this applet will be something like:

```
Initialised -- Starting -- Painting.
```

As you will note the methods stop() and destroy() are not overridden, because they are not needed by this simple applet.

Awt class does have 2 more methods update() and repaint().

6.13. Update and repaint methods of AWT class:

Awt class does have 2 more methods update() and repaint().

6.13.1 update()

It is important in some situations your applet may need to overide the update() method. This method is defined by the AWT and is called when your applet has requested that a portion of its window be redrawn. The problem is that the default version of update() first fills an applet with the default background colour and then calls paint(), this gives rise to a flash of default color (usually gray) each time update is called. To avoid this you define your update() method such that it performs all necessary display activities [NOTE: if you define your own update() method then the default update() is not called - this is called overriding]. The paint() in this casewill simly call update(). Take a look at the following piece of code for better understanding.

```
public void update(Graphic g) {
    //Redisplay your window here.
}

public void paint(Graphics g) {
    update(g) // call to the update()method.
}
```
You need to override update () only when needed.

6.13.2 Requesting Repainting!

An applet writes to its window only when its update() or paint() methods are called by the AWT. You must be wondering how can the applet, itself, cause its window to be updated when its information changes? For example, if an applet is displaying a moving banner, what mechanism does the applet use to update the window each time this banner scrolls? Remember, you cannot create a loop inside paint due the fundamental contraints imposed on an applet - An applet must quickly return control to the AWT run time system. Fortunately the people who designed Java foresaw this predicament and included the repaint() method. Whenever your applet needs to update the information displayed in its window, it simply calls repaint().
The repaint() method is defined by the AWT. It causes the AWT run time system to call to your applet's update() method, which in its default implementation, calls paint(). Again for example if a part of your applet needs to output a string, it can store this string in a variable and then call repaint(). Inside paint(), you can output the string using drawstring().
The repaint method has four forms.
```
void repaint()
void repaint(int left, int top, int width, int height)
void repaint(long maxDelay)
void repaint(long maxDelay, int x, int y, int width, int height)
```

Let's look at each one -
void repaint()
This causes the entire window to be repainted

void repaint(int left, int top, int width, int height)
This specifies a region that will be repainted. the integers left, top, width and hieght are in pixels. You save time by specifying a region to repaint instead of the whole window.

void repaint(long maxDelay)
void repaint(long maxDelay, int x, int y, int width, int height)
Calling repaint() is essentially a request that your applet be repainted sometime soon. However, if your system is slow or busy, update() might not be called immediately. This gives rise to a problem of update() being called sporadically. If your task requires consistent update time, like in animation, then use the above two forms of repaint(). Here, the maxDelay() is the maximum number of milliseconds that can elaspe before update() is called.

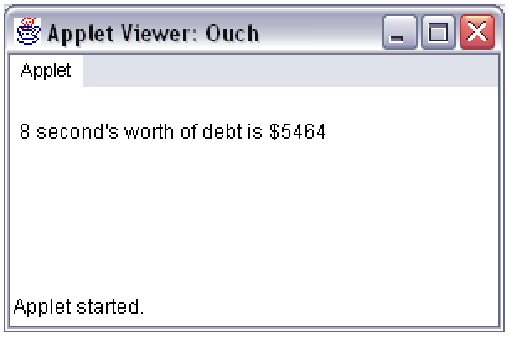
6.13.3 Difference between paint() and repaint()

The paint() method contains instructions for painting the specific component. We cannot call this method directly instead we can call **repaint()**.

The repaint() method, which can't be overridden, is more specific: it controlsthe update() to paint() process. You should call this method if you want a componentto repaint itself or to change its look (but not the size)

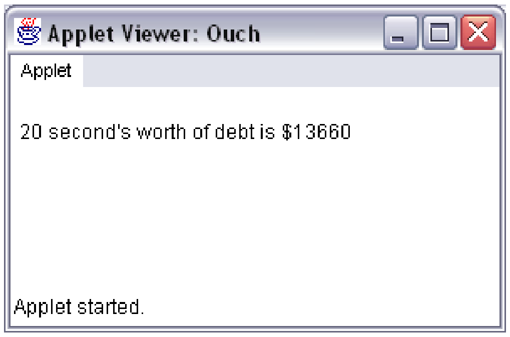
Example1:

```
import java.awt.*;
import java.util.*;
public class Ex_repaint extends java.applet.Applet
{
    int debt = 683;
    int totalTime = 1;
     public void paint(Graphics g)
        {
        g.drawString(totalTime + " second's worth of debt is $" + (debt * totalTime), 5, 30);
        for (int i = 0; i < 250000000; i++);
        totalTime++;
        repaint();//calls paint() again
        }
 }
```



```
8 second's worth of debt is $5464
```

```
Applet started.
```



```
20 second's worth of debt is $13660
```

```
Applet started.
```

6.14 The HTML APPLET Tag

The APPLET tag is used to start an applet from both an HTML document and from an applet viewer. An applet viewer will execute each APPLET tag that it finds in a separate window,
While web browsers like Netscape Navigator, Internet Explorer, and HotJava will allow many applets on a single page.

The syntax for the standard APPLET tag is shown here. Bracketed items are optional.

< APPLET
[CODEBASE = *codebaseURL*]
CODE = *appletFile*
[ALT = *alternateText*]
[NAME = *appletInstanceName*]
WIDTH = *pixels* HEIGHT = *pixels*
[ALIGN = *alignment*]
[VSPACE = *pixels*] [HSPACE = *pixels*]
>
[< PARAM NAME = *AttributeName* VALUE = *AttributeValue*>]
[< PARAM NAME = *AttributeName2* VALUE = *AttributeValue*>]
. . .
[*HTML Displayed in the absence of Java*]
</APPLET>

Let's take a look at each part now.
CODEBASE: CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag). The HTML document's URL directory is used as the CODEBASE if this attribute is not specified. The CODEBASE does not have to be on the host from which the HTML document was read.

CODE: CODE is a required attribute that gives the name of the file containing your applet's compiled **.class** file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.

ALT: The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser understands the APPLET tag but can't currently run Java applets. This is distinct from the alternate HTML you provide for browsers that don't support applets.

WIDTH AND HEIGHT: WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area.

ALIGN: ALIGN is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

VSPACE AND HSPACE: These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes.

PARAM NAME AND VALUE: The PARAM tag allows you to specify appletspecific arguments in an HTML page. Applets access their attributes with the **getParameter( )** method.

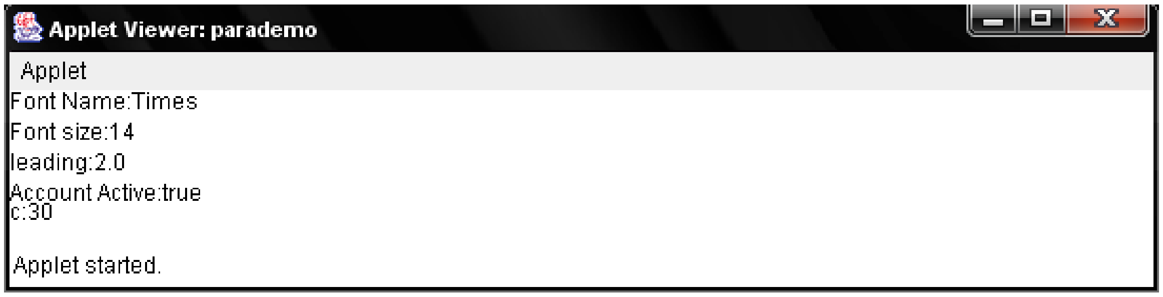
6.15 Passing parameters to Applets

Applet html tag allows you to pass parameters to your applet. To retrieve a parameter, use the getParameter() method. It returns the value of the specified parameter in the form of a String object.

Example:

```
import java.awt.*;
import java.applet.*;
/*<applet code = "parademo" width = 600 height=80>
<param name=fname value=Times New Roman>
<param name=fsize value=14>
<param name=lead value=2>
<param name=accountEnabled value=true>
</applet>*/

public class parademo extends Applet
{
String fname;
int  fsize;
float lead;
boolean active;
int a,b,c;
        public void start()
        {
        String param;

        fname=getParameter("fname");
        if(fname==null)
        fname="not found";

                param = getParameter("fsize");
                try
                {
                        if(param!=null)
                        fsize=Integer.parseInt(param);
                        else
                        fsize=0;
                }
                catch(Exception e)
                {
                fsize= -1;
                }

                param=getParameter("lead");
                try
                {
                        if(param != null)
                        lead=Float.valueOf(param).floatValue();
                        else
                        lead=0;
                }
                catch(Exception e)
                {
                lead= -1;
                }
        param=getParameter("accountEnabled");
        if(param!=null)
        active=Boolean.valueOf(param).booleanValue();

        a=10;
        b=20;
        c=a+b;

        }//start

public void paint(Graphics g)
{
        g.drawString("Font Name:"+fname,0,10);
        g.drawString("Font size:"+fsize,0,26);
        g.drawString("leading:"+lead,0,42);
        g.drawString("Account Active:"+active,0,58);
        g.drawString("c:"+c,0,68);
}//paint

}//parademo
```

In given applet program, getParameter method returns values in String object so for numeric, float & boolean values , we convert their string representation into their internal formats.



```
Font Name:Times
Font size:14
leading:2.0
Account Active:true
c:30

Applet started.
```

6.16 Applet Methods

There are different types of methods for the Graphics class of the *java.awt.*;* package has been used to draw the appropriate shape.

1) setBackground
public void setBackground(Color c)
Sets the background color of this component.
The setBackground() method is defined by the Component class. It is used to set the background of an applet's window to a specified color. Here, new color specifies the color. The Color class defines the constants, which can be used to set the color. Some of them are listed below:
Color.black(pink,green,yellow,blue,magenta,orange,red,white,lightGray,darkGray,gray,cyan)
The background color affects each component differently and the parts of the component that are affected by the background color may differ between operating systems.
Parameters:
c - the color to become this component's color; if this parameter is null, then this component will inherit the background color of its parent

2)setForeground
public void setForeground(Color c)
Sets the foreground color of this component.
Parameters:
c - the color to become this component's foreground color; if this parameter is null then this component will inherit the foreground color of its parent

3)getBackground
public Color **getBackground**()
Gets the background color of this component.
Returns: this component's background color; if this component does not have a background color, the background color of its parent is returned

4)getForeground
public Color getForeground()
Gets the foreground color of this component.
Returns: this component's foreground color; if this component does not have a foreground color, the foreground color of its parent is returned

5)setColor
This is the setColor() method which is the Graphics class method imported by the *java.awt.*;* package. This method sets the color for the object by specified color. Here is the syntax of the setColor() method :
g.setColor(**Color.**color_name);

6)drawString
The drawSring() method draws the given string as the parameter on particular position using graphical interface.
void drawString(String string, int X_coordinate, int Y_coordinate);
Text is displayed in an applet window by using the drawString() method of the Graphics class.The drawString() method is similar in function to the System.out.println() method that displays information to the system's standard output device. Before you can use the drawString() method, you must have a Graphics object that represents the applet window.
The paint() method of all applets includes a Graphics object as its only argument. This object represents the applet window, so it can be used to create a Graphics object that also represents the window.
The following three arguments are sent to drawString():

- The text to display, which can be several different strings and variables pasted together with the + operator
- The x position (in an (x,y) coordinate system) where the string should be displayed
- The y position where the string should be displayed

Ex: g.drawString("hello",10,10);

7)drawLine
The drawLine() method has been used in the program to draw the line in the applet.
Here is the syntax for the drawLine() method :
drawLine(int X_from_coordinate, int Y_from_coordinate, int X_to_coordinate, int Y_to_coordinate);
Ex: g.drawLine(3,300,200,10);

8)drawOval
The drawOval() method draws the circle. Here is the syntax of the drawOval() method :
g.drawOval(int X_coordinate, int Y_coordinate, int Wdth, int height);
Ex: g.drawOval(10,10,50,50);

9)drawRect
The drawRect() method draws the rectangle. Here is the syntax of thedrawRect() method :
g.drawRect(int X_coordinate, int Y_coordinate, int Wdth, int height)
Ex: g.drawRect(10,10,50,50);

10)fillOval
This is the fillOval() method used to fill the color inside the oval by specified color. Here is the syntax of the fillOval() method :
g.fillOval(int X_coordinate, int Y_coordinate, int Wdth, int height);
Ex: g.drawOval(10,10,50,50);

11)fillRect
This is the fillRect() method used to fill the color inside the rectangle by specified color. Here is the syntax of the fillRect() method :
g.fillRect(int X_coordinate, int Y_coordinate, int Wdth, int height)
Ex: g.fillRect(10,10,50,50);

12)drawRoundRect & fillRoundRect
It draws an outlined round-cornered rectangle. Here are the syntaxies and examples of both the functions.
Syntax: drawRoundRect(int x,int y,int width,int height,int arcWidth,int arcHeight)
Ex: g.drawRoundRect(200,300,100,100,50,50);
Syntax: fillRoundRect(int x,int y,int width,int height,int arcWidth,int arcHeight)
Ex: g.fillRoundRect(200,300,100,100,50,50);

Example 1:
```
import java.applet.*;
import java.awt.*;
public class ShapColor extends Applet{
  int x=300,y=100,r=50;
  public void paint(Graphics g){
  g.setColor(Color.red); //Drawing line color is red
  g.drawLine(3,300,200,10);
  g.setColor(Color.magenta);
  g.drawString("Line",100,100);
  g.drawOval(x-r,y-r,100,100);
  g.setColor(Color.yellow); //Fill the yellow color in circle
  g.fillOval( x-r,y-r, 100, 100 );
  g.setColor(Color.magenta);
  g.drawString("Circle",275,100);
  g.drawRect(400,50,200,100);
  g.setColor(Color.yellow); //Fill the yellow color in rectangel
  g.fillRect( 400, 50, 200, 100 );
  g.setColor(Color.magenta);
  g.drawString("Rectangel",450,100);
  }
}
```

Example 2:

```
/*<applet code =shapes width=500 height=500></applet>*/
import java.applet.*;
import java.awt.*;
public class shapes extends Applet{
  int x=300,y=100;
  public void paint(Graphics g){
           g.drawLine(3,300,200,10);
           g.drawString("Line",100,100);

           g.drawOval(x,y,100,100);
           g.setColor(Color.cyan);
           g.fillOval(x+1,y+1,99,99);
           g.setColor(Color.black);
           g.drawString("Circle",325,150);

           g.drawRect(400,50,100,100);
           g.setColor(Color.red);
           g.fillRect(401,51,99,99);
           g.setColor(Color.black);
           g.drawString("Rectangel",420,100);

           g.fillRoundRect(200,300,100,100,50,50);
           g.setColor(Color.blue);
           g.drawRoundRect(200,300,100,100,50,50);
           g.drawString("RoundRect",220,350);
  }
}
```

13)Drawing Polygons

Polygons are shapes with many sides. A polygons may be defined as a set of connected lines. The end of first line is the beginning of second line, and so on,

Syntax:

drawPolygon(int[] xPoints, int[] yPoints, int nPoints)

Draws a closed polygon defined by arrays of x and y coordinates.

Example:

```
import java.awt.*;
import java.applet.*;
public class Poly extends Applet
{
```

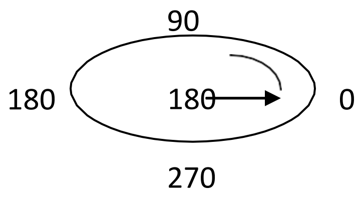
```
int x1[]={20,120,220,20};
int y1[]={20,120,20,20};
int n1=4;
int x2[]={120,220,220,120};
int y2[]={120,20,220,120};
int n2=4;
public void paint(Graphics g)
{
g.drawPolygon(x1,y1,n1);
g.fillPolygon(x2,y2,n2);
}
}
```

14)Drawing Arc

drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)

Draws the outline of a circular or elliptical arc covering the specified rectangle.

Java considers 3 O'clock as 0 degree position and degree increases in **anti-clock** wise direction.

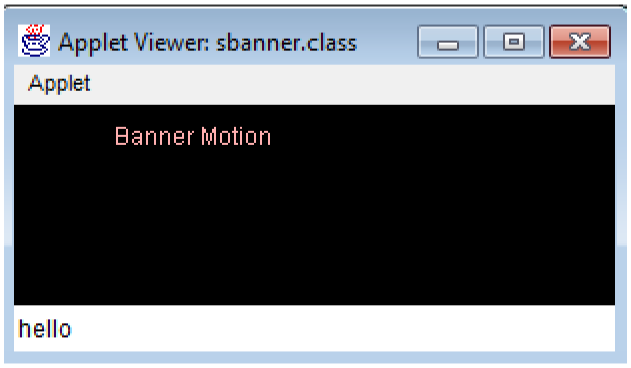


Example:

```
import java.awt.*;
import java.applet.*;
public class Face extends Applet
{
public void paint(Graphics g)
{
g.drawOval(40,40,120,150); //Head
g.drawOval(57,75,30,20); //Left eye
g.drawOval(110,75,30,20); //Right eye
g.fillOval(68,81,10,10); //Pupil (left)
g.fillOval(121,81,10,10); //Pupil (right)
g.drawOval(85,100,30,30); //Nose
g.fillArc(60,125,80,40,180,180); //Mouth
g.drawOval(25,92,15,30); //Left ear
g.drawOval(160,92,15,30); //Right ear
}
}
```

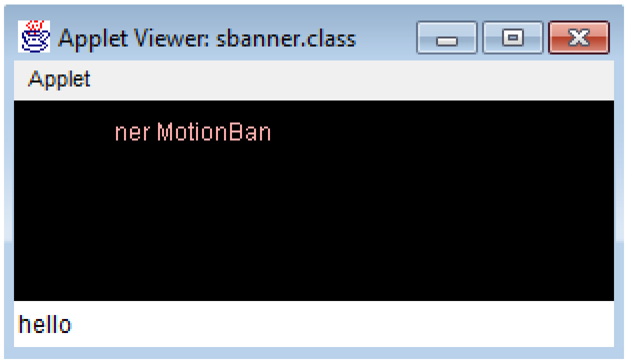
Program: Message movement program at fix location using repaint and thread.

```
import java.awt.*;
import java.io.*;
import java.applet.*;
import java.lang.Thread;

/* <applet code = "sbanner.class" width = 300 height=100></applet> */

public class sbanner extends Applet implements Runnable{
String msg= "Banner Motion";
Thread t = null;
int state;
boolean stopF;

public void init()
{
setBackground(Color.black);
setForeground(Color.pink);
stopF = false;
}//init

public void start()
{
t=new Thread(this);
t.start();
}//start

public void run()
{
char c;
for(; ;)
{
try
{
repaint();
Thread.sleep(500);
c=msg.charAt(0);
msg=msg.substring(1,msg.length());
msg+=c;
//System.out.println(msg);
if(stopF)
break;
}
catch(InterruptedException e)
{}
}//for
}//run
public void stop()
{
stopF=true;
t=null;
}//stop

public void paint(Graphics g)
{
g.drawString(msg,50,20);
showStatus("hello");
}//paint
}//sbanner
```



```
Banner Motion
```

```
hello
```



```
ner MotionBan
```

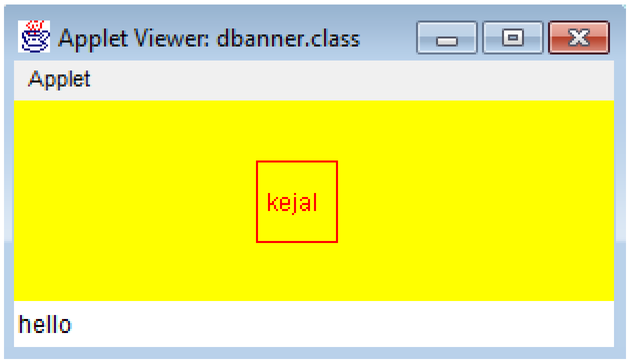
```
hello
```

Program: Movement of Object with a Message towards left hand Side.

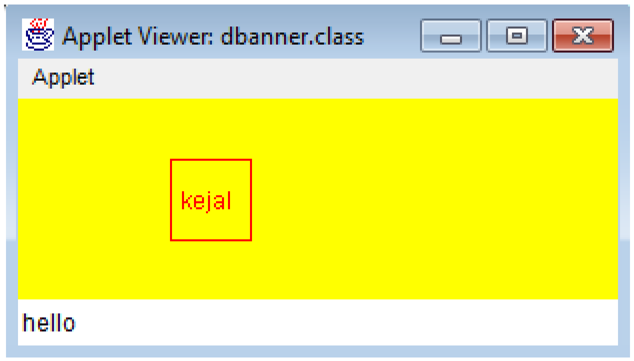
```
import java.awt.*;
import java.io.*;
import java.applet.*;
import java.lang.Thread;

/* <applet code = " dbanner.class" width = 300 height=100></applet> */

public class dbanner extends Applet implements Runnable{
Thread t = null;
```

```
boolean stopF;
int width, height;
int x, y;
public void init()
{
    width = getSize().width;
    height = getSize().height;
    setBackground( Color.yellow);
    x = width/2 - 20;
    y = height/2 - 20;
    stopF = false;
}//init

public void start()
{
t=new Thread(this);
t.start();
}//start

public void run()
{
for(; ;)
{
try
{
Thread.sleep(250);
repaint();
x=x-1;
if(stopF)
break;
}
catch(InterruptedException e)
{}
}//for
}//run

public void paint(Graphics g)
{
setForeground( Color.red );
g.drawRect( x, y, 40, 40 );
g.drawString("kejal",x+5,y+25);
showStatus("hello");
}//paint
```

```
public void stop()
{
stopF=true;
t=null;
}//stop
}//class
```



```
hello
```



```
hello
```

6.17 Exercise

1. How mutex can be achieved in java?
2. What is the purpose of yield() method?
3. What is daemon thread?
4. Explain the life cycle of thread.
5. Write a short note on:
    1) Thread synchronization.
    2) Inter-thread communication.
    3) Deadlock Condition
6. What is an applet? How to pass parameters to the applet? Explain with example.
7. Explain applet life cycle also explain any three methods of graphics class.
8. Write down difference between Applet and Application.

7. Practical Exercise

1. Write a program to print the area and perimeter of a shape triangle having sides of 3 by creating a class named 'Triangle' without any parameter in its constructor.
2. Write program to calculate area of square and circle using method overloading
3. Write a program to input at least 5 records of teaching staff and 5 records of non-teaching staff and display it using inheritance.
4. Write a program to perform multiple inheritance using interface
5. Write a program that creates two interfaces 1. Direction 2. Drive Car. And creates two classes 1. Direction Board 2. Car which inherits above interfaces.
6. Write a program to demonstrate partial implementation of interface and extending interfaces.
7. Write a package that shows all combinations of the access control modifiers.
8. Write a program to accept an array & count total no of invalid entries & display valid entries using user define exception.
   Input: 1 2 3 -1 (invalid) 6 -8(invalid)
   Output: 1 2 3 6
9. Check whether given string is palindrome or not.
10. Count total no. of vowel, words, char & special characters
11. Write a program to sort or arrange the names in alphabetical order.
    Input: Reena , Ankit, Hema
    Output: Ankit, Hema, Reena
12. Write a program to generate menu driven program for getChars, insert, replace, reverse, setCharAt, deleteCharAt, Toggle Case, Upper case & lower Case.
13. Create the dept-emp mgt system using concept of nested class.
14. Write a program to implement stack using thread
15. Write a program to accept multiple line contents until you press "N" Then check total no of consonants, lines & total no. of word.
    Input:
    Enter Strings:
    This is java book
    Continue: Y
    Java is programming Lang.
    Continue: N
    Output:
    Total line: 2
    Consonants: 22
    Words: 08
16. Create 2 classes which contains following details.

Emp table	**Dept table**
Empno	Deptno
Empname	Deptname
Esal	

Edesignation
Enter atleast 5 records & display report in format & Accept data through user & print it to console.

Ename	Edesignation	Deptname	Esal

17. Write a program which has 2 applet. The 1st accepts name & print in second applet & 2nd applet accept address & print it in 1st applet.
18. Write a program to move square or triangle from one end to another end.
19. Write a program add/sub/mul/div of 2 numbers in Applet.
20. Accept value in 2 textboxes & create button name “+”,”-“,”/”,”*”, when you click them, it will give result in third box.
21. Write a program to print even and odd number at interval of 1 second.
22. Write a program to accept string and print total no. of vowels and consonants.
23. Write a program to accept numbers from command line and display sum of odd and even numbers.
24. Write a program to run car from left to right. Write car name on car.
25. Write a program to accept 2 numbers from command line and create 2 threads to display odd and even numbers between these 2 numbers.
26. Write a program to accept 10 names of students and their age. Sort name and age in descending order. Display the name of student using thread at interval of 1 second.
27. Write an applet to print rectangle using color and print name inside it.
28. Write a program which accepts starting and ending character from string. Display each character between then at interval of 1 second.
29. Write an applet which accepts age as parameter and display message as “my age is: ___” and if age parameter not passed then take default age as 45.
30. Write a program to accept number from command line and check whether it is palindrome or not.
31. Write a program to accept number from command line and display individual digit at interval of 3 second.
32. Write an applet to accept string as parameter and display odd character in red and even character in green color.
33. Write an applet program to display digital clock with date and time at center of applet using different colors.
34. Write an applet program to add 2 numbers using textbox and button.
35. Write a program to accept string from command line and display each character in reverse at interval of 1 second.
36. Write an applet to display smiley with appropriate greeting message.
37. Write a program which accepts 2 string arguments or 2 number arguments from command line. Write an overloading method which will concate string or add number arguments.
38. Design item class item_code, item_name, item_price and stock in hand. Write a java program which accepts N item and display item detail in ascending order of stock.
39. Write an applet to accept no. of oval user want to display in parameter tag and drawthat many ovals in different positions.

40. Write an applet to draw solid square using red color. Write college name within square with font style “time new roman”, font size “14” and font color “yellow”.
41. Write a program to accept string and arrange them in alphabetical order.
    Input: computer
    Output: cemoprtu.
42. Write a program to design string class which performs different methods like equals, reverse, change case and trim.
43. Write a program to create an interface. Create 3 class rectangle, circle and triangle and calculate their area respectively.
44. Write an applet to draw triangle in circle. Fill them with different colors using parameter tag.
45. Write an applet program to print name in following pictures & fill different colors in picture.