

CHAPTER1

INTRODUCTION TO JAVA

Q) What is java?

- Java is more than a programming language called as Technology.
- JAVA Mainly three things
 1. Object oriented programming language
 2. Platform
 3. Technology
- Java is just a small, simple, secure, portable, object-oriented, interpreted, byte coded, architectural-neutral(platform independent), garbage collected, multithreaded programming language with strongly typed exception handling mechanism for writing distributed, dynamically extensible programs.

Q) What is a programming language?

A **programming language** is an artificial language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs that control the behavior of a machine and/or to express algorithms precisely.

A programming language is usually split into the two components of syntax (form) and semantics (meaning). Some languages are defined by a specification document (for example, the C programming language is specified by an ISO Standard).

Q) What is java Technology?

Technology: Technology is the practical application of knowledge so that something entirely new can be done, or so that something can be done in a completely new way. So it is the way of solving problems through the application of knowledge from multiple disciplines.

- Java technology is object oriented platform independent multithreaded programming environment.
- Object oriented programming environment means that support the following concept.
 1. Class
 2. Object
 3. Encapsulation
 4. Inheritance
 5. Polymorphism
 6. Abstraction
 7. Message passing
 8. Dynamic Binding

Q) Why java is so popular?

- In real time environment projects are mainly divided into three types. They are
 1. Desktop projects
 2. Enterprise web projects
 3. Device projects
- Java is so popular because by using java we can develop any kind (above three types) of project.
- For desktop projects java platform standard edition (J2SE) is used.
- For enterprise web projects java platform enterprise edition (J2EE) is used.
- To develop device projects java platform micro edition (J2ME) is used.

Q) Who created the java?

- “**James gosling**” sun Microsystems created java in 1991
- But released in 1996
- Original name of java is oak. But there was a programming language already with the name oak, so Mr. James Gosling and his team decided to change the programming language name to java.

Q) What is original motivation of java technology?

- Original motivation of java technology is to create platform independent programming language that can be used to develop programs to operate consumer electronics devices.

Examples: refrigerators, TV set boxes, micro ovens, washing machines Etc.

- But later the original motivation of java technology changed to create platform independent programming language that can be used to develop secured network applications.

Q) What is an internet application?

An application that can be executed from remote computer via network call (request) is called as internet application. That remote computer can have any type of OS.

Types of internet applications:

We have two types of internet applications

1. web supportive applications
2. web application

Web supportive applications:

An application that resides in server system and if it is downloaded and executed in client computer via network call, then that application is called **Web supportive applications**

Web application:

An application that resides in server system and if it is executed directly in server system via network call, then that application is called web application

Q) Explain about Java Features.

Java is the language that has become most successful and popular because of the following features.

1) Platform Independent:

->The java programs compiled on one operating system can be transferred and executed on any Operating System without modifications. This can be achieved through an application called "Java Virtual Machine" or simply JVM. This can be achieved as follows:

->Once we create the java source code, we compile it using Java Compiler. The compiler then creates something called Java Byte code. This byte code can be copied and executed anywhere i.e., even in mobile phones also. Thus, Java is also called "Architecture Neutral" Language.

2) Object-Oriented:

-> Java is the only Language that is a pure object-oriented. That is, every concept of OOP is supported by Java. Also, it is noted that even the main () function should be defined under a class.

3) Compiled and Interpreted:

-> Java is a language which provides both compilation and interpretation of programs. Once java program is created, it is compiled by Java Compiler. This compiled code (Byte code) can be executed by using Java Interpreter.

4) Multi-Threaded:

-> Using this feature, Java supports "Multitasking". Multitasking is the concept of executing multiple jobs simultaneously. Multitasking improves the CPU and Main Memory Utilization.

5) Dynamic:

->This is also one of the important features that made Java popular. Assume that in a program, we created 100 functions. In no case, all the functions are executed. But, in languages like C, whether required or not, all the functions are loaded into memory which result in wastage of memory.

->But, in Java, until you call a function, it is not loaded into memory. The functions are loaded only when they are called (i.e. at run-time).

6) Simple, small and familiar:

-> Java programs are easy to build and implement when compared to languages like 'C' and 'C++' because most of the concepts in these languages which people felt difficult and confusing are eliminated in Java. Also, the concepts in C and C++ which programmers felt comfortable are included in Java.

-> Java is a small programming language and therefore it requires less memory and less time to load and install. Because of its sophisticated features, Java has become familiar.

7) Robust and Secure:

-> As a robust language, Java provides many safeguards to ensure reliable code. It also provides exception handling concept to handle logical errors that make a system to crash.

-> As a secured language, Java ensures that programs cannot gain access to memory locations without proper authorization.

8) Distributed:

-> Java is a distributed language for creating networking applications. It has the ability to share both data and programs.

-> Java applications provide mechanisms to open and access objects remotely.

9) High Performance:

-> Java architecture is designed to reduce overheads during run-time. Also, the concept of multithreading in Java increases the execution speed of Java Programs.

Q) What is current version of java?

Java 7 is the current version. Even though it is released currently java 5 and java 6 versions are used in the industry.

List of java versions:

JDK VERSION	RELEASED IN	CODE NAME
1.0	January, 1996	-
1.1	February, 1997	Pumpkin
1.2(java 2)	December, 1998	Play ground
1.3(java 2)	May, 2000	Kestrel
1.4(java 2)	February, 2002	Merlin
Java 5	September, 2003	Tiger
Java 6	December, 2006	Mustang
Java 7	July 28, 2011	Dolphin

Q) What is source code and Executable code?

Source code: Group of instructions which are present in high level language is known as source code. Generally source code visible in English

Executable code: Group of instructions which are present in low level language is known as Executable code. Generally Executable code visible in machine language format

Q) What is platform?

Ans: Operating system and Architecture of the processor together is known as platform.

Q) What is Java platform? How it is classified?

Execution environment or runtime infrastructure for the execution of Java application is known as Java Platform.

Java platform is divided into 3 categories

1. J2SE Platform
2. J2EE Platform
3. J2ME platform

Q) What do you mean by platform independent and platform dependent with respect to programming language?

- If the compiled code of a program could be executed irrespective of the operating system under which it has been compiled, then we call that program and its programming language as platform independent.
- If the compiled code of a program could not be executed irrespective of the operating system under which it has been compiled, then we call that program and its programming language as platform dependent.

Q) What is the Need for Platform Independence?

It is the concept of executing the compiled code of a program created on one operating system onto another. The main reason for developing Java programming Language is Platform Independence.

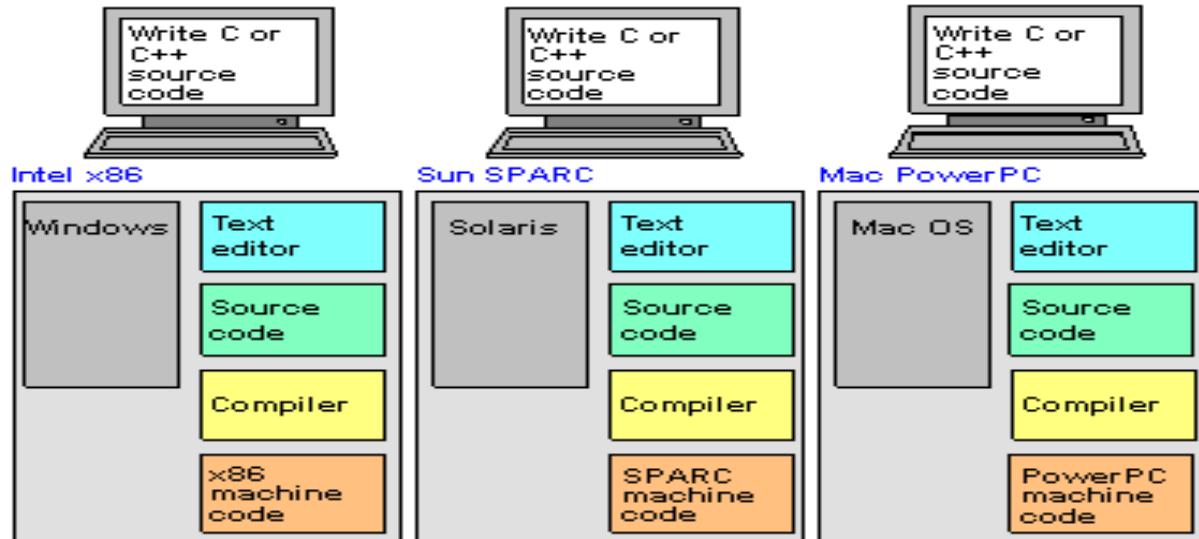
->"Source Code" is the intellectual property for any Software Development Organization. This means that no IT development company will provide its source code to its users because of "piracy problems". Hence, they have to provide compiled code of software to their users.

-> But, the problem here is the users must have the same Operating Environment that the IT Company has. This may not be possible in all cases because, the customers and developers may have different operating environments in their organizations. If the customers are having different operating environments, the software companies have to reject their request. This problem made software development companies develop a new Programming Language "Java".

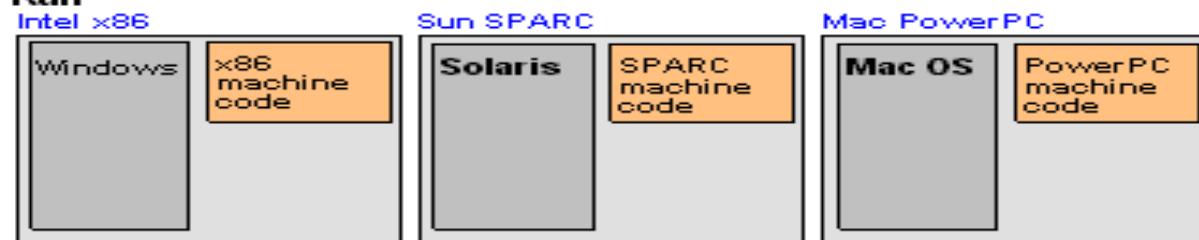
→ In the below diagram we are going to observe the exact need of platform independent applications.

From Computer Desktop Encyclopedia
© 2003 The Computer Language Co., Inc.

Create & Modify in C



Run



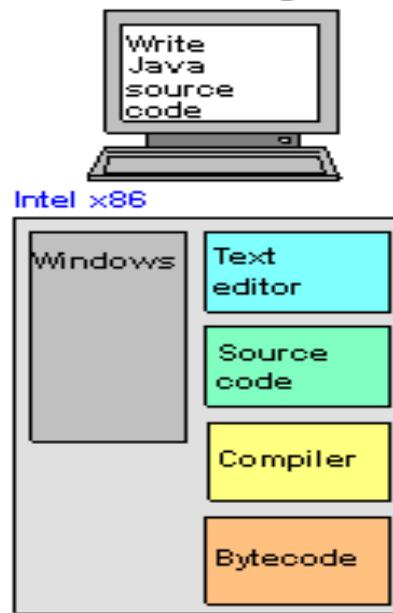
Q) How Java is platform independent?

- Compile code of java is the byte code. Byte code could be executed irrespective of the operating system under which it has been compiled. So java is platform independent.
- If you observe the below the diagram we will get more clarity about platform independent applications.

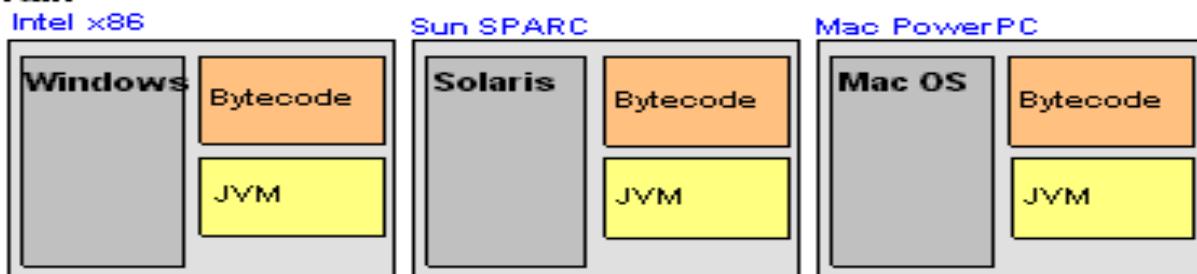
- If you are compiling our java code on windows operating system. Then it generate one intermediate code called as "BYTE CODE".
- But at run time you are giving the same byte code for different operating systems, i.e. windows, Solaris, Mac Os, all operating systems are accepted and generated its respective machine code by using JVM

From Computer Desktop Encyclopedia
© 2003 The Computer Language Co., Inc.

Create & Modify in Java



Run



Java Uses an Intermediate Language

Java source code is compiled into an intermediate language called "byte code." The byte code can be run in any hardware that has a Java Virtual Machine (JVM) for that machine platform. Thus, the "**write once-run anywhere**" concept

Q) Is the JVM is platform independent or dependent?

Ans: JVM is platform dependent.

Q) What is JVM and how its converts byte code into machine code?

- JVM is Software from Sun that converts a program in Java from byte code (intermediate language) into machine language and executes it. The Java Virtual Machine (JVM) is the runtime engine of the Java Platform, which allows any program written in Java or other language compiled into Java byte code to run on any computer that has a native JVM. JVMs run in both clients and servers, and the Web browser can activate the JVM when it encounters a Java applet.
- The JVM includes a just-in-time (JIT) compiler that converts the byte code into machine language so that it runs as fast as a native executable. The compiled program can be cached in the computer for reuse.

- JVM is an interpreter and it converts .class (dot class) file code into operating system code.
- JVM converts .class file code line by line and executes line by line.
- If 1st line of .class file code is converted into operating system code then it is executed at that time.
- Similarly for second, third and fourth nth lines its process should be the same.

Q) What is compiler?

A **compiler** is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code). The most common reason for wanting to transform source code is to create an executable program.

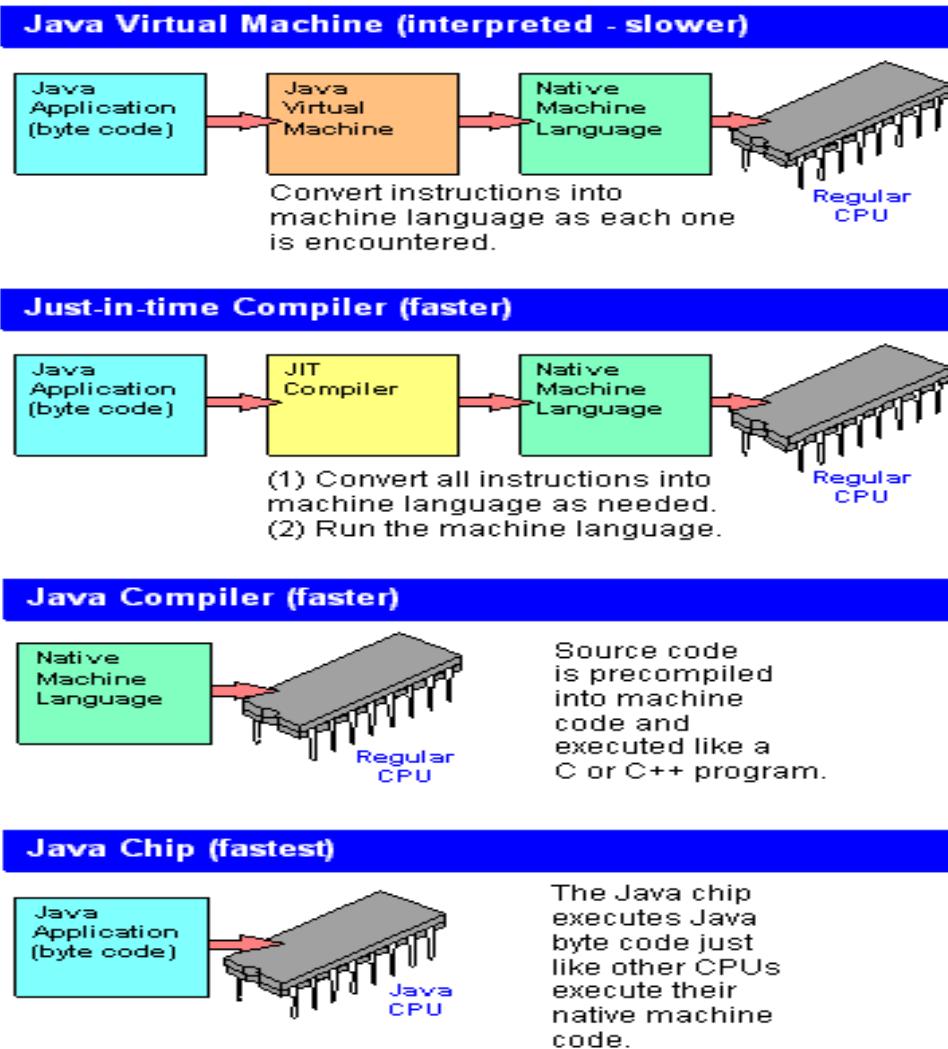
The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code). If the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is known as a cross-compiler. A program that translates from a low level language to a higher level one is a decompiler. A program that translates between high-level languages is usually called a language translator, source to source translator, or language converter. A language rewriter is usually a program that translates the form of expressions without a change of language.

Program faults caused by incorrect compiler behavior can be very difficult to track down and work around; therefore, compiler implementors invest a lot of time ensuring the correctness of their software.

Q) What is JIT?

(**Just-In-Time compiler**) A compiler that converts program source code into native machine code just before the program is run. In the case of Java, a JIT compiler converts Java's intermediate language (bytecode) into native machine code as needed. It tries to predict which instructions will be executed next so that it can compile the code in advance. Compiled code resides in memory until the application is closed.

From Computer Desktop Encyclopedia
© 2004 The Computer Language Co. Inc.



Q) Why JVM code is called as Byte code?

- JVM code is called Byte Code because JVM language mnemonics are 256.
- To store any mnemonics we need minimum of one byte. So JVM mnemonics are called as Byte code.

Q) What is Byte Code?

The source code of a Java program is compiled into an intermediate language called "byte code." In order to run the byte code, it must be compiled into machine code just before execution or a line at a time via the Java Virtual Machine (JVM) runtime engine. There are JVMs for all major hardware platforms (i.e. for all operating systems).

Q) Why Java interpreter is called as JVM?

- Java interpreter works similar to actual micro processor (machine).
- Java interpreter is working same as actual machine so it is called virtual machine (means not a real machine). Micro processor is a hardware component.

- JVM is not hardware component, but it is software component.

Q) Who does develop JVM, OS vender or SUN?

JVM is developed by SUN not by OS vendor.

Q) Will JVM available for all operating systems?

Yes JVM is available for all operating systems separately.

Q) Why does java use both compiler and interpreter?

Java is both compiled and interpreted language. First Java source code has to be translated into Byte code, which is done with the help of a compiler. But these byte codes are not machine instructions. Therefore ,in second stage this byte code has to be translated into machine code. This task is performed by an Interpreter. Hence, Java use both compiler and interpreter.

Q) What are the primary goals of java technology?

There are four primary goals of java technology

1. Solve the problems with existing programming languages such as C and C++
 - I. Security problem
 - II. Memory leak
 - III. Unable to handle run time errors
 - IV. Unable to develop pure object oriented programs
2. Write once and run anywhere(platform indecency or portability)
3. Multithreading
4. Dynamic loading

Q) Who will de allocate the memory in java?

Ans: Garbage collector

Q) What is garbage collector?

Ans: Garbage collector is a software tool that is used to de allocates memory that is not in use. Garbage collector is a part of Java runtime environment (JRE)

Q) What is Java Runtime Environment?

Ans: It is an Environment in which java programs are executed. JRE is a part of JDK.

Q) Explain about JDK (JAVA development kit).

- All the programs which are required for developing a java application, compiling a java program, debugging and executing a java application and the library packages and class would present in JDK.

- Thus JDK is a package or folder which contains

1. Programs required for developing java application

2. Java compilers

3. Java debuggers

4. JVM

5. Library classes and Library packages

- JVM would be a part of JDK. We can download JDK or we can download only JVM.
- If the compiled code of a program could be executed irrespective of the operating system under which it has been compiled, then we call that program and its programming language as platform independent.
- Java developers require JDK for developing java applications. But whereas a client requires only Jvm.
- The size of the JDK would be more than the size of the JVM.

Q) What is loading? And why loading?

- .exe files, .bin files, .class files are executable files containing machine or jvm code.
- All these files are stored in hard disk.
- To execute all these files by microprocessor these files from hard disk must be loaded into ram
- To load .exe files, .bin files the operating system uses static loading.
- In static loading all executable blocks are loaded into ram if all executable blocks loading is completed then execution of the program is started
- JVM loads .class files dynamically, so .class files loading is dynamic loading
- In dynamic loading 1st main class is loaded, then main function execution is started
- Based on main function lines execution if required other classes also loaded
- All classes in JVM are not loaded at the beginning of main function execution.
- Whenever main function is under execution at that time remaining classes are loaded based on requirements.

Environment variables:

PATH: Path is an environmental variable using which we make JDK and its programs available to the local operating systems. Path of the JDK has to be mentioned in the environment variable path.

Using this environment variable path local operating system would be able to locate the programs belonging to JDK.

Ex: path: jdk\bin

CLASSPATH:

CLASSPATH is also an environment variable using which we make all the library packages and the required programs available to the JVM. Thus JVM would be locating the library packages through the environment variables CLASSPATH

Ex: jdk\lib\tools.jar;jdk\jre\lib\rt.jar

NOTE:

1. Environment variable path would be used by local operating system and environment variable class path would be used by JVM.
2. All the basic library packages which are required for developing a basic java application are present in "rt.jar" file. In the form of a compressed format.

CHAPTER2

BASIC OBJECT ORIENTED CONCEPTS

OBJECT ORIENTED PROGRAMMING:

Q) What is object oriented programming language?

Object-orientation is a set of tools and methods that enable software engineers to build reliable, user friendly, maintainable, well documented, reusable software. System that fulfills

the requirements of its users. It is claimed that object-orientation provides software developers with new mind tools to use in solving a wide variety of problems. Object-orientation provides a new view of computation. a software system is seen as a community of objects that cooperate with each other by passing messages in solving a problem.

An object-oriented programming language provides support for the following object-oriented concepts:

- objects and classes
- inheritance
- polymorphism and dynamic binding

Q) What are the Principles of object oriented programming?

Object orientation is of the programming styles or methodologies. As far as application development is concerned, following are the important object oriented features.

1. Encapsulation
2. Inheritance
3. Polymorphism

Encapsulation:

The concept of **binding** the data along with its related and corresponding functionalities or functions is known as encapsulation.

The concept of making the data available only to **certain** related areas or only within mentioned borders is known as binding.

Inheritance:

A key feature of JAVA classes is inheritance. Inheritance allows to create classes which are derived from other classes, so that they automatically include some of its "parent's" members, plus its own.

The concept of getting the **properties** of one class to another class is known as inheritance.

Polymorphism:

Poly means many and morphism means forms (functionalities), so the concept of defining multiple functionalities or methods with the same name associated with the same object is known as polymorphism.

Q) Give any four advantages of OOPS.

1. The principle of data hiding helps the programmer to build secure programs That cannot be invaded by code in other parts of the program.
2. It is possible to have multiple instances of an object to co-exist without any Interference
3. Object oriented programming can be easily upgraded from small to large systems.

4. Software complexity can be easily managed.

Q) Give any four applications of OOPS

- Real-time systems.
- Simulation and modeling.
- Object-oriented databases.
- AI and expert systems.

Q) What Is Class?

- Class is nothing but a structure binding the data along with its corresponding and related functions.
- Class is a user defined data type in java. Class will act as the base for encapsulation.
- Class is a similar (but not exactly) to "structure" in c programming language to create user defined data types that model real world entities.
- Class is not an object oriented feature. But class is the means with which all object oriented features are programmatically realized. For example, encapsulation, polymorphism and inheritance can't be programmatically achieved without class concept. But it does not mean that class is an object oriented feature.
- A class contain properties and operations
- A class is a plan for the proposed object.
- A class is known as a blueprint or template for an object.
- Defining a class is nothing but modeling a real world entity. Real world entity type creation is nothing but classification. That is why the name "class" which is nothing but type.
- Any java application works like collection of classes.

Q) What is general form class definition?

General form of a class definition:

```
class classname {  
    type instance-variable1;  
    type instance-variable2;  
    // ...  
    type instance-variableN;  
    type methodname1(parameter-list) {  
        // body of method  
    }  
}
```

```
type methodname2(parameter-list) {  
    // body of method  
}  
  
// ...  
  
type methodnameN(parameter-list) {  
    // body of method  
}  
  
}
```

Q) What is object?

- Physical realization of a class is nothing but an object.
- Instantiating a class is nothing but creating an object.
- A class is the basic of encapsulation. An object implements encapsulation.
- From class any number of objects can be created.
- Object is nothing but some memory area in RAM to store data in a secured manner.
- Fundamental unit of data storage in an object oriented system to store data in a secured manner is nothing but an object.

An object is associated with 3 things:

1. State
 2. Behavior
 3. Identity
- Properties/variable/attributes and data stored at that time in those variables put together is nothing but state of the object. I.e. data stored in an object is known as object state.
 - Operations/methods of the object are nothing but object behavior.
 - Name of the object with which it is uniquely identified is nothing but identify of an object.

Q) What is the need of object?

Ans: In order to allocate memory space and load the members of a class to the RAM from the byte code at the Run time we need object.

Q) Explain Creating Objects in Java?

-> We know that a class is a collection of variables and the methods that access those variables. Generally, when a program starts execution, the JVM first loads the main() method

into memory and starts execution. Because the main() is residing in memory, the remaining members of the class are not available to main(). To make the remaining members of a class(non-static) available to main(), we create objects. That is ,an Object is the memory created for the members (non-static) of a class. The procedure for creating an object is as follows:

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

class-name reference;

- Second, acquire an actual, physical copy of the object and assign it to that variable using the new operator.

reference = new class-name();

- The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it.
- This reference is, more or less, the address in memory of the object allocated by new.
- This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.
- At this point, you might be wondering why you do not need to use new for such things as integers or characters.
- The answer is that Java's simple types are not implemented as objects.
- Rather, they are implemented as "normal" variables. This is done in the interest of efficiency.
- It is important to understand that new allocates memory for an object during run time.
- The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program.

For example, for a class Sample, we can create an object as follows :

Sample s;

s = new Sample();

->We can also create an object in a single statement as follows:

class-name reference = new class-name();

Eg : Sample s = new Sample();

-> Once the object is created, we can access the class members in main() using the dot(.) operator.

Q) What object contains?

CALL:9010990285/7893636382

Ans: Object of any class contains only data. Object of a class would not contain the logics or the functionalities of a function.

Q) If you modify one object data will another object data also be modified?

No, modifications done for one object will not be affected to another object.

Q) Till what period of time object of a class would be persistent?

Ans: As long as address of the object is persistent, the object is also persistent in the ram. Once address of the object is loss, Automatically the corresponding object deleted from the ram.

Q) When object is available to a method?

Ans: whenever we make a method call with an object and if the functionality of a method is loaded to the ram because of an object, then we can says that object is available for that method.

All the non-static members of a class would be directly available to every non-static method of the same class.

Q) What are the instance variable?

Ans: Any non static variable of a class can be called as an instance variable. All the non static variables of a class would be loaded to the ram through an instance of class. And there all present inside of a class thus these variables are known as instance variable.

Q) What is the difference between a reference variable and pointer variable in java?

Ans: A pointer variable contains the actual address but where as a reference variable contains only the index of the address.

The actual address of an object would be maintained by the jvm internally and that can be seen by the end user. Thus people say that java doesn't support pointers. But java support pointers, that pointer are called as restricted pointers. Restricted pointers mean pointers without arithmetic support.

Q) What is dynamic Binding?

Linking the statements of a method when it is called, is called "Binding". If the statements of a method are linked at the time of program compilation, it is called "Static Binding" (E.g.: C-Language). If the statements of a method are linked at the time of program execution, it is called "Dynamic Binding" (E.g.: OO languages).

Q) What is message passing?

Objects communicate with each other by passing messages between them which we call message passing. Message Passing involves:

1. creating classes

2. creating object
 3. establishing link with objects
- E.g.: b. deposit (SB2045,20000);

Q) What is Data Abstraction?

The major goal of OOP is "Data Security". This can be achieved by hiding variables of a class from external functions. This concept of hiding the data members of a class from outside functions is called "Data Abstraction".

Q) What are different types of inner classes?

A) **Nested top-level classes**- If you declare a class within a class and specify the static modifier, the compiler treats the class just like any other top-level class. Any class outside the declaring class accesses the nested class with the declaring class name acting similarly to a package. e.g., outer. Inner. Top-level inner classes implicitly have access only to static variables. There can also be inner interfaces. All of these are of the nested top-level variety.

Member classes - Member inner classes are just like other member methods and member variables and access to the member class is restricted, just like methods and variables. This means public member class acts similarly to a nested top-level class. The primary difference between member classes and nested top-level classes is that member classes have access to the specific instance of the enclosing class.

Local classes - Local classes are like local variables, specific to a block of code. Their visibility is only within the block of their declaration. In order for the class to be useful beyond the declaration block, it would need to implement a more publicly available interface. Because local classes are not members the modifiers public, protected, private and static are not usable.

Anonymous classes - Anonymous inner classes extend local inner classes one level further. As anonymous classes have no name, you cannot provide a constructor.

Inner class inside method cannot have static members or blocks

Q) How to declare classes, methods and attributes in java?

Declaring Classes:

```
<Access_Specifier> class <Class_name> {
    <attribute declaration>;
    <constructor declaration>;
```

```
<Method declaration>  
}
```

Declaring Attributes:

```
<Access_Specifier> <type> <name>=<initial-value>;
```

Declaring Methods:

```
<Access_Specifier> <return_type> <name>(<argument>){  
    <statements>  
}
```

Q) What is the difference between procedural and object-oriented programs?

1. In procedural program, programming logic follows certain procedures and the instructions are executed one after another. In OOP program, unit of program is object, which is nothing but combination of data and code.
2. In procedural program, data is exposed to the whole program whereas in OOPs program, it is accessible within the object and which in turn assures the security of the code.

Q) Differentiate between the message and method.

Message

- * Objects communicate by sending messages to each other.
- * A message is sent to invoke a method.

Method

- * Provides response to a message
- * It is an implementation of an operation.

Q) How many bits are used to represent Unicode, ASCII, UTF-16, and UTF-8 characters?

Unicode requires 16 bits and ASCII require 7 bits. Although the ASCII character set uses only 7 bits, it is usually represented as 8 bits. UTF-8 represents characters using 8, 16, and 18 bit patterns. UTF-16 uses 16-bit and larger bit patterns..

Q) Is size of a keyword?

The size of operator is not a keyword..

Q) Does garbage collection guarantee that a program will not run out of memory?

Garbage collection does not guarantee that a program will not run out of memory. It is possible for programs to use up memory resources faster than they are garbage collected. It is also possible for programs to create objects that are not subject to garbage collection.

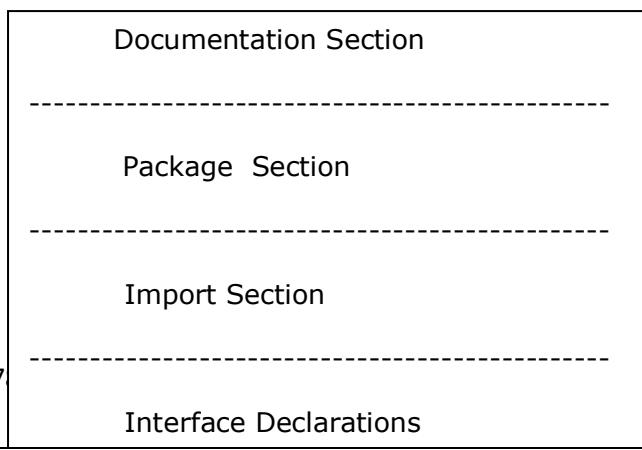
JAVA BY SATEESH

CHAPTER3

STRUCTURE OF A JAVA PROGRAM AND JAVA MODIFIER

Structure of a Java Program:

To create a program in Java, we have to follow the following structure:



In the above structure, any comments about the program should be declared in the "Documentation Section".

-> We learnt that Java supports a set of built-in packages. If we want to create our own package with a set of classes, we have to use "Package Section". If we want to use any class defined under a package, we have to import that package to our program using "Import Section".

-> Java supports 3 types of structures namely interfaces, classes and abstract classes. The interfaces, if required, should be declared under "Interface Declaration Section".

-> If the program contains a set of classes, they are placed under "Class declaration Section". Java is a pure Object-Oriented Language. This means, every function in a Java program must be encapsulated. That is, even the main () method must be declared under some class.

Example :

```
1  /** This is a Sample program in Java */
2  class Sample
3  {
4      public static void main(String args[])
5      {
6          System.out.println ("This is My First Java Program");
7      }
8  }
```

-> In the above program, Line-1 indicates a comment about our program. The word "Sample" is the name of the class containing main () method. In Line-4, "public" is the keyword that makes main () accessible globally. The keyword "static" ensures that main () is accessible even though that class contains no objects. The keyword "void" indicates that main () method does not return any value. In Line-4, String args[] represents the set of String objects passed as parameters to main() from command line. To produce the output on the console, we use "System.out.println()" method. This statement is explained as below.

Explanation of statement System.out.println():

-> println() is an overloaded method defined in java.io.PrintStream class. The reference of this class is created as a static member in the java.io.System class. Through this reference, we are calling System.out.println() method

Public static void main (String [] args):

Q) What if the main method is declared as private?

The program compiles properly but at runtime it will give "Main method not public." Message

Q) What if the static modifier is removed from the signature of the main method?

Program compiles. But at runtime throws an error "NoSuchMethodError".

Q) Can you write "static public void" instead of "public static void" but not "public void static".

Yes, We can write "**static public void**" instead of "**public static void**" but not "**public void static**".

Q) If i do not provide the String array as the argument to the method?

Program compiles but throws a runtime error "NoSuchMethodError".

Q) If no arguments on the command line, String array of Main method will be empty or null?

It is empty. But not null.

Q) Can an application have multiple classes having main method?

A) Yes it is possible. While starting the application we mention the class name to be run. The JVM will look for the Main method only in the class whose name you have mentioned. Hence there is not conflict amongst the multiple classes having main method.

Q) Can I have multiple main methods in the same class?

A) No the program fails to compile. The compiler says that the main method is already defined in the class.

Practise Examples:

// Write a program to Add Two Integers

```
package BASICS.CORE;
public class Sum {
    public static void main(String[] args) {
        int a=5;
        int b=10;
```

```
int c=a+b;  
System.out.println("Addition of two numbers:"+c);  
}  
}  
// Write A Program, Add Two Integers By Taking Values At Run Time  
package BASICS.CORE;  
import java.io.BufferedReader;  
import java.io.InputStreamReader;  
public class InputRuntime2 {  
    public static void main(String[] args) throws Exception {  
        InputStreamReader isr = new InputStreamReader(System.in);  
        BufferedReader br = new BufferedReader(isr);  
        System.out.println("Enter The First Number:");  
        String s1 = br.readLine();  
        System.out.println("Enter The Second Number:");  
        String s2 = br.readLine();  
        int a = Integer.parseInt(s1);  
        int b = Integer.parseInt(s2);  
        int c = a+b;  
        System.out.println("Sum Of Two Numbers is:"+c);  
        br.close();  
        isr.close();  
    }  
}
```

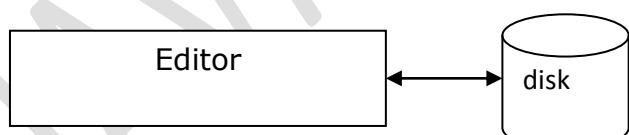
```
// Write A Program, Add Two Integers By Taking Values At Run Time  
package BASICS.CORE;  
import java.util.Scanner;  
public class RunInput {
```

```
public static void main(String[] args) {  
    Scanner sc = new Scanner(System.in);  
    System.out.println("Enter The First Number:");  
    int a = sc.nextInt();  
    System.out.println("Enter The Second Number:");  
    int b = sc.nextInt();  
    int c = a+b;  
    System.out.println("Sum Of Two Number Is:"+c);  
}  
}  
  
// write a program to call one function with in another function
```

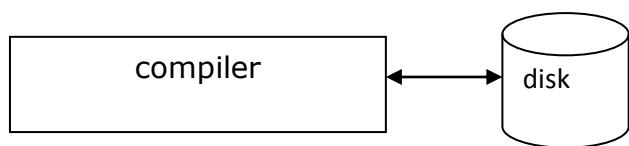
```
package BASICS.CORE;  
public class NestFunCalling {  
    static void fun1(int a ){  
        System.out.println("Function1 Begins");  
        fun2(a);  
        System.out.println("In Side Function1:"+a);  
    }  
    static void fun2(int x){  
        System.out.println("In Side Function2:"+x);  
    }  
    public static void main(String[] args) {  
        System.out.println("Main Started");  
        int y=10;  
        fun1(y);  
        System.out.println("In Side Main:"+y);  
        System.out.println("Main Ended");  
    }  
}
```

}

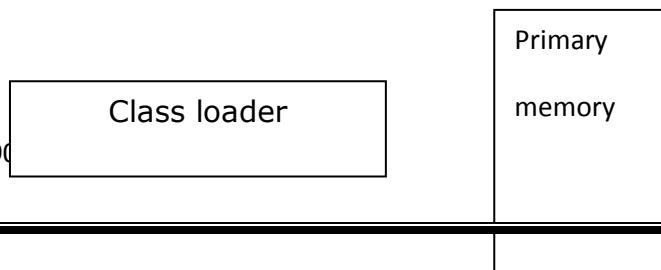
Conclusion:



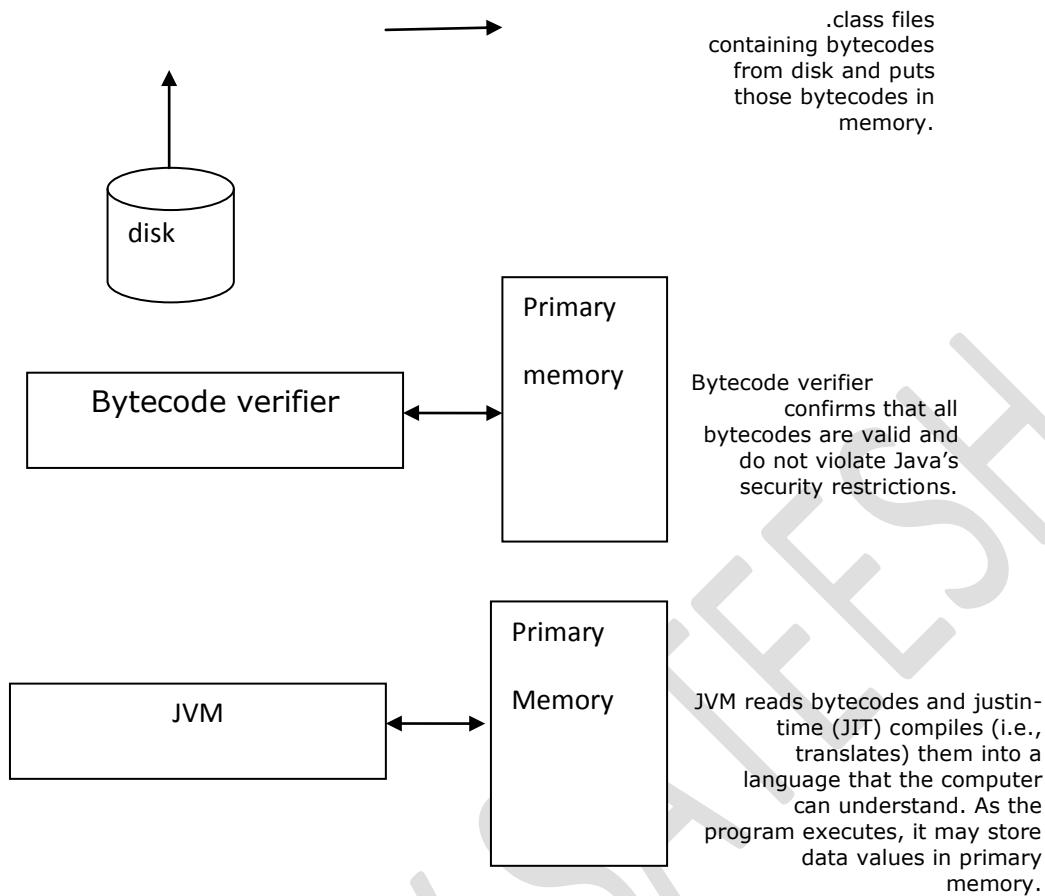
Program is created in an
Editor and stored on disk in
A file ending with **.java**



Compiler creates bytecodes
And stores them on disk in a
file ending with **.class**.



CALL:90



Java Modifier Types:

Modifiers are keywords that you add to those definitions to change their meanings. The Java language has a wide variety of modifiers, including the following:

- Java Access Modifiers
- Non Access Modifiers

To use a modifier, you include its keyword in the definition of a class, method, or variable. The modifier precedes the rest of the statement, as in the following examples

```

public class className {
    // ...
}
private boolean myFlag;

static final double weeks = 9.5;
    
```

```
protected static final int BOXWIDTH = 42;

public static void main(String[] arguments) {
    // body of method
}
```

Access Control Modifiers:

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

Visible to the package. The default, No modifiers are needed.

->This means that a java element is not declared as public or private or protected. This default specifier:

-> Will act as public for the same class members or other classes that are in the same path.

-> will act as private for the classes that are defined outside the path

-> will act as protected for the derived classes of same path and outside the path

Visible to the class only (private)

Any java element that is declared as private can be accessible to the members of same class only. It cannot be available for the classes that are in the same path or outside the path.

Visible to the world (public)

Any java element that is declared as public can be accessible to the members of same class or the classes in the same directory or the classes in another path.

Visible to the package and all subclasses (protected)

Any java element that is declared as protected can be accessible to the same class members or to its sub-classes within the same path or outside the path.

Non Access Modifiers:

Java provides a number of non-access modifiers to achieve many other functionality.

- The static modifier for creating class methods and variables
- The final modifier for finalizing the implementations of classes, methods, and variables.
- The abstract modifier for creating abstract classes and methods.
- The synchronized and volatile modifiers, which are used for threads.

Conclusion:

	Private	Public	Protected	No modifier
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	Yes	Yes	No
Different package non-subclass	No	Yes	No	NO

CHAPTER4

JAVA TOKENS

Tokens in Java

-> A token is the smallest individual element used to create a program statement. Java supports the following types of tokens:

- 1) Reserved Words.
- 2) Identifiers.
- 3) Literals.
- 4) Operators.
- 5) Special Symbols.

1) Reserved Words: They are also called "keywords" which are having some special purpose and meaning. These words are to be used for that purpose only.

-> Java supports the following set of reserved words.

abstract	assert	boolean	break
byte	case	catch	char
class	const	continue	default

do	double	else	enum
extends	final	finally	float
for	goto	if	implements
import	instanceof	int	interface
long	native	new	package
private	protected	public	return
short	static	strictfp	super
switch	synchronized	this	throw
throws	transient	try	void
volatile	while		

2) Identifiers:

- > They are also called "user-defined words". These are names for program elements like variables, constants, functions, interfaces, packages, abstract classes, classes and objects.
- > To create an identifier, we have to follow the following rules.

1. First letter should be alphabet.
2. All alphabets (A-Z, a-z), digits (0-9) are allowed.
3. Special symbols (except '_') are not allowed.
4. Spaces are not allowed.
5. Reserved words are not allowed. Etc.,

Naming Conventions:

Constants ->All letters are capitals.

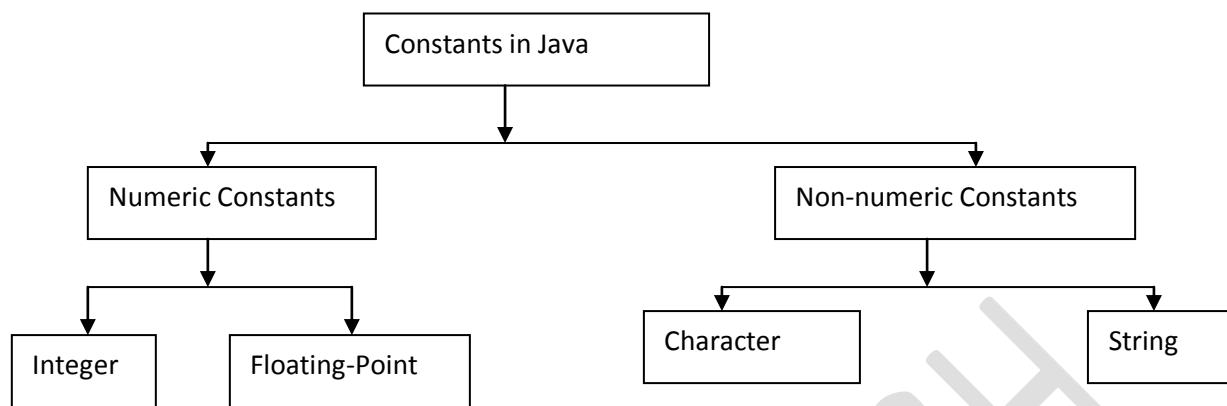
Variables -> All letters are small letters.

Classes and other structures -> First letter of every word is capital.

Functions -> First letter of every word, except first word is capital.

3. Java Literals:

- > Literals are also called "constants". A constant is a value that does not change during program execution.
- > Java supports the following constants.



A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Literals can be assigned to any primitive type variable. For example:

```
byte a = 68;
char a = 'A'
```

byte, int, long, and short can be expressed in decimal(base 10),hexadecimal(base 16) or octal(base 8) number systems as well.

Prefix 0 is used to indicates octal and prefix 0x indicates hexadecimal when using these number systems for literals. For example:

```
int decimal = 100;
int octal = 0144;
int hexa = 0x64;
```

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are:

```
"Hello World"
"two\nlines"
"\\"This is in quotes\\""
```

String and char types of literals can contain any Unicode characters. For example:

```
char a = '\u0001';
String a = "\u0001";
```

Java language supports few special escape sequences for String and char literals as well. They are:

Notation	Character represented
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\f	Formfeed (0x0c)
\b	Backspace (0x08)
\s	Space (0x20)
\t	tab
\"	Double quote
'	Single quote
\\	backslash
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)

4) Operators:

-> An operator is a special symbol used in expressions. Java supports the following set of operators:

a) Arithmetic Operators:

These operators are used for Arithmetic operations. These operators include: +,-,*/,%

b) Relational Operators:

They are also called ' Comparison operators' because they are used to compare two values. These include: ==, !=, >, >=, <, <=.

c) Logical Operators:

They are also called 'Boolean operators' because they return either 'true' or 'false' values. They include:

-> Logical AND (&&) -> this operator returns 'true' if all the conditions are 'true'. Otherwise, it returns 'false'.

-> Logical OR (||) -> this operator returns 'true' if at least one condition is 'true'. It returns 'false' if all the conditions are 'false'.

-> **Logical NOT (!)** -> This operator reverses the value of a condition. i.e., it returns 'true' if the condition is 'false', and returns 'false', if the condition is 'true'.

d) Unary operators :

These operators are used on 'single' operands. They include:

Unary plus (+) -> represents a positive value.

Unary minus (-) -> represents a negative value.

Increment (++) -> It increments the value of a variable by 1. This operator is used in two ways. They are pre-increment (++i) and post-increment (i++) .

Decrement (--) -> these operator decrements the value of a variable by 1. This operator is used in two ways. They are pre-decrement (--i) and post-decrement (i--) .

e) Assignment operator :(=)

-> This operator is used to store a value into a variable. The syntax of assignment is:
variable = value | variable | expression;

Eg: a = 10; -> value assignment.

a = b; -> variable assignment.

a = b + c; -> expression assignment

-> Along with this symbol(=) , we can also use the following assignment operators.

Eg: a = a + 1; <=> a += 1; Also, a = a + b; <=> a += b;

Like this, we can use -=, *=, /=, %= operators.

f) Conditional Operator (?, :)

-> This is also called "Ternary operator" because this operator is used on 3 operands. Its syntax is :

(condition)?(true-part):(false-part);

Eg: if(a>b) c=a; else c=b;

is same as

c= (a>b)?a:b;

g) Bitwise Operators :

-> Like C and C++, Java supports a set of Bitwise operators to manipulate bits. They include:

Bitwise AND &

Bitwise OR !

Bitwise XOR ^

1's complement	~
left shift	<<
right shift	>>

h) Special Operators:

- **dot (.)** :- This operator is used in accessing class members through reference.
- **new**: Dynamic memory allocation.
- **instanceof:=** This operator returns 'true' if an object is instance of a class. Otherwise, it returns 'false'. Its syntax is :

object instanceof class-name

5) Special Symbols:

a) Comments:

Comments in a program can be used for documentation purpose and user understanding. Java supports the following comment symbols.

C -> /* */ -> Block comments

C++ -> //..... -> Line comments

Java -> /** */ -> Block Comments

b) Separators:

Comma -> value-separator

Semi-colon -> End of statement

c) White Space characters:

-> Blank space, tab space, enter key.

d) Brackets:

{ } -> represent blocks, used for array initialization

() -> represent functions, conditions

[] -> arrays

Comments in Java:

Java supports single line and multi-line comments very similar to c and c++. All characters available inside any comment are ignored by Java compiler.

```
public class MyFirstJavaProgram{
```

```
    /* This is my first java program.
```

```
* This will print 'Hello World' as the output
* This is an example of multi-line comments.
*/
public static void main(String []args){
    // This is an example of single line comment
    /* This is also an example of single line comment. */
    System.out.println("Hello World");
}
}
```

JAVA VARIABLE:

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

A class can contain any of the following variable types.

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

Source file declaration rules:

Let us now look into the source file declaration rules. These rules are essential when declaring classes, import statements and package statements in a source file.

- There can be only one public class per source file.
- A source file can have multiple non-public classes.

- The public class name should be the name of the source file as well which should be appended by **.java** at the end. For example : The class name is . public class Employee{} Then the source file should be as Employee.java.
- If the class is defined inside a package, then the package statement should be the first statement in the source file.
- If import statements are present then they must be written between the package statement and the class declaration. If there are no package statements then the import statement should be the first line in the source file.
- Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file.

Classes have several access levels and there are different types of classes; abstract classes, final classes etc.

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

Q) What is the use of an import statement?

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

Example:

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

JAVA BY SATEESH

CHAPTER4

DATA MEMBERS

Types of data members:

In java we have two types of data members. They are

1. Static data member
2. Non static data members

Q) What does static mean?

static - stationary, standing, not moving

Q) What is static data member?

- Static data member is a data member declared with a static data modifier.
- Static data members are created inside class memory (static memory).
- Static data members is not created for every object it is created only once in class memory and shared by all objects.

Q) What are the types of static members?

Java support four types of static members

1. static variables
2. static blocks
3. static methods
4. main method

Q) What is the execution order of all static variables?

Ans: static variable is a class level variable which is declared by using static keyword. All the static variables are executed by jvm in the order they are defined from top to botttom.

Note: All static variables are having individual memory locations.

Q) How to access static data member?

- In three ways we can access static data members
 1. Classname.datamember
 2. Reference.datamember(here reference can point null)
 3. Reference.datamember(here reference can point object)

Q) What is non - static data member?

- Non Static data member is a data member declared without a static data modifier.
- Non Static data members are created inside object memory
- Non static data members are created for every object.

Q) What are the types of Non Static members?

Java supports four types of non static members

1. Non static variables
2. Non static blocks
3. Non static methods
4. constructors

Q) When do all these members get memory locations and by whom?

CALL:9010990285/7893636382

All non static data members get memory location only if object is created with new keyword and constructor of that class. JVM will not provide memory location for these members by default by itself.

Q) How to access Non static data member?

- In two ways we can access non static data members
 1. Reference.datamember(here reference must point object, if reference points null then NullPointerException is raised)
 2. new ClassName().datamember

Q) How can we call non static members from other non static members?

We can call non static members from other non static members directly by their name.

Ex:

```
class MCA
{
    int rno=100;
    float avg=78.7f;
    void first()
    {
        System.out.println("I am in first");
    }
    void second()
    {
        System.out.println(rno);
        System.out.println(avg);
        first();
    }
}
public static void main(String[] args) {
    MCA m=new MCA();
    m.second();
}
```

Q) Why we cannot call Non static members from static members directly?

Because static member can be accessed directly without object creation, hence compiler doesn't allow non static members directly from static members.

Examples on static and non static data members:

//Write a java program to call static with classname

```
class MyStatic{
    static int a;
}
```

```
class MyStaticTest{
```

CALL:9010990285/7893636382

```
public static void main(String[] args){  
    MyStatic.a=10;  
    System.out.println("My Static value is:"+MyStatic.a);  
}  
}
```

Output:

My Static value is:10

Example2:

// Write a java program to verify three to access static data members

```
class MyStatic{  
    static int a;  
}  
  
class MyStaticTest1{  
    public static void main(String[] args){  
        MyStatic s1,s2;  
        s1=new MyStatic();  
        s2=null;  
        MyStatic.a=10;  
        System.out.println("My Static value is:"+MyStatic.a);  
        s1.a=20;  
        System.out.println("My Static value is:"+s1.a);  
        s2.a=30;  
        System.out.println("My Static value is:"+s2.a);  
    }  
}
```

Output:

My Static value is:10

My Static value is:20

My Static value is:30

Example3:

```
// All static member are stored in class memory

class MyStatic{

static int a;

}

class MyStaticTest1{

public static void main(String[] args){

MyStatic s1,s2;

s1=new MyStatic();

s2=null;

MyStatic.a=10;

s1.a=20;

s2.a=30;

System.out.println("My Static value is:"+s1.a);

System.out.println("My Static value is:"+MyStatic.a);

System.out.println("My Static value is:"+s2.a);

}

}
```

Output:

```
My Static value is:30
My Static value is:30
My Static value is:30
```

Example4:

```
// write a program to call non static member with a class

class MyStatic{

int a;

}

class MyStaticTest{
```

```
public static void main(String[] args){  
    MyStatic.a=10;  
    System.out.println("My Static value is:"+MyStatic.a);  
}  
}  
}
```

Output:

```
MyStaticTest.java:6: non-static variable a cannot be referenced from a static co  
ntext
```

```
MyStatic.a=10;
```

```
^
```

```
MyStaticTest.java:7: non-static variable a cannot be referenced from a static co  
ntext
```

```
System.out.println("My Static value is:"+MyStatic.a);
```

```
^
```

2 errors

Example5:

// Write a program to access Non static data members

```
class MyStatic{  
    int a;  
}  
  
class MyStaticTest{  
    public static void main(String[] args){  
        MyStatic s1 = new MyStatic();  
        s1.a=10;  
        System.out.println("My Static value is:"+s1.a);  
    }  
}
```

Output:

My Static value is:10

Example6:

CALL:9010990285/7893636382

//Write a java to verify can you call non static data members with a reference

```
class MyStatic{
    int a;
}

class MyStaticTest{
    public static void main(String[] args){
        MyStatic s1 = new MyStatic();
        MyStatic s2 = null;
        s1.a=10;
        System.out.println("My Static value is:"+s1.a);
        s2.a=10;
        System.out.println("My Static value is:"+s2.a);
    }
}
```

Output:

My Static value is:10

Exception in thread "main" java.lang.NullPointerException

at MyStaticTest.main(MyStaticTest.java:10)

Difference between static data members and non static data members:

Static data members	Non Static data members
1. static data members are declared with static modifier.	1. Non static data members are declared without static modifier
2. Static data members are created in class memory	2. Non Static data members are created in object memory
3. We can access static data members by using class name or by using reference pointing object or pointing to null.	3. We can access non static members only using object or object pointing to reference.
4. Static data is common for all objects	4. Non static data is specific for each and every object
5. Static variables are called class	5. Non static variables are called as

variables.	instance variables or object variables.
6. Static data can be access in both static and non static methods.	6. Non static data can be access only in non static methods.

Methods In Java

- In C language a big program can be divided into small modules called as functions.
- Where as in java a big program can be divided into small modules called as classes.
- In java class contain “data members” and “methods”.
- Data members are used to hold the and methods are used to perform operations on the data defined in the class.

Q) What is a method?

Methods - A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.

General Structure of a class:

```
class Bank{
    int accno;
    float bal;
    void saving(){
    }
    Void deposit(){
    }
    Void loan(){
    }
}
```

Syntax to define methods in java:

```
[Modifiers] <return-type> method name ([parameter list]){
    // Body of the method
}
```

Explanations:

A method definition consists of a method header and a method body. Here are all the parts of a method:

Modifiers: The modifier, which is optional, tells the compiler how to call the method. This defines the access type of the method.

Modifiers can be public, private, protected, static, final, abstract, strictfp, synchronized and annotations

Return Type: A method may return a value. The returnType is the data type of the value the method returns. Some methods perform the desired operations without returning a value. In this case, the returnType is the keyword **void**.

Method Name: This is the actual name of the method. The method name and the parameter list together constitute the method signature.

Parameters: A parameter is like a placeholder. When a method is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.

Method Body: The method body contains a collection of statements that define what the method does.

Types of methods:

In java we have two types of methods

1. Static methods
2. Non static methods

Static methods:

- Static method is declared with static modifier.
- Static method can be called by using

ClassName.methodName

Or

Reference.methodName (Here reference points null)

Or

Reference.methodName (Here reference not points null)

Q) Will jvm executes static methods by default like static variables?

Jvm will not execute static methods by itself. They are executed only if they are called explicitly by developer either from main method or from static variable as its assignment statement or from static block.

Q) What is need of static method?

If you need to access a method without creating an object of corresponding class, it need to be a static method.

Q) What is the order of execution of static methods?

Static methods are executed in the order they are called, not in the order they are defined.

Q) Difference between instance method and static method in java?

Instance methods can be called by the object of a Class whereas static method are called by the Class.

When objects of a Class are created, they have their own copy of instance methods and variables, stored in different memory locations.

Static Methods and variables are shared among all the objects of the Class, stored in one fixed location in memory. Static methods **cannot** access instance variables or instance methods directly-they must use an object reference. Also, class methods cannot use the this keyword as there is no instance for this to refer to.

Q) Static method can be overridden?

Yes, static methods can be overridden just like any other methods.

Q) What does it mean when a method or field is static?

A static field or method is a keyword to indicate that:

- A) For a class member, the memory address of the member for all instances of this class is shared.
- B) For a class method, an instance of this class is not needed to be called.

Examples:

// A class static method can be called in the same class without class name and without ref name

```
public class Demo {  
  
    public static void main(String[] args) {  
        fun(); // with out class name and without ref name  
        Demo.fun(); // with class name  
        Demo r1,r2;  
        r1=new Demo();  
        r2 = null;  
        r1.fun(); // with ref name(ref doesn't assign to null)  
        r2.fun(); // with ref name but ref assigned to null  
    }  
}
```

```

    }

    static void fun()
    {
        System.out.println("I Am In Fun");
    }
}

```

// main method can call only static methods of the same class because main is also static method

```

class Sample {
    void fun1(){
        System.out.println(" I Am in Fun1");
    }
    static void fun2(){
        System.out.println(" I Am in Fun2");
    }
    public static void main(String[] args) {
        //fun1(); it leads to compilation error
        fun2();
    }
}

```

Q) What is Non static method?

The method which doesn't have static keyword in its prototype is called non static method. Like non static variables, non static methods also should be called only by using object.

Q) How to call Non static method?

reference.methodname();

or

new ClassName().methodname();

Example:

```

package NONSTATIC;

public class NonStaticDemo {
    void myself()
    {
        System.out.println("I am In Fun");
    }
    public static void main(String[] args) {

```

```
// myself(); Not allowed
new NonStaticDemo().myself();
NonStaticDemo s = new NonStaticDemo();
s.myself();

}
```

JAVA BY SATEESH

CHAPTER 5

DATA TYPES

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java:

1. Primitive Data Types
2. Reference/Object Data Types

Primitive Data Types:

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a key word. Let us now look into detail about the eight primitive data types.

byte:

- Byte data type is a 8-bit signed two's complement integer.
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive) ($2^7 - 1$)
- Default value is 0

- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.
- Example : byte a = 100 , byte b = -50

short:

- Short data type is a 16-bit signed two's complement integer.
- Minimum value is -32,768 (- 2^{15})
- Maximum value is 32,767(inclusive) ($2^{15} - 1$)
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an int
- Default value is 0.
- Example : short s= 10000 , short r = -20000

int:

- Int data type is a 32-bit signed two's complement integer.
- Minimum value is - 2,147,483,648.(- 2^{31})
- Maximum value is 2,147,483,647(inclusive).($2^{31} - 1$)
- Int is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0.
- Example : int a = 100000, int b = -200000

long:

- Long data type is a 64-bit signed two's complement integer.
- Minimum value is -9,223,372,036,854,775,808.(- 2^{63})
- Maximum value is 9,223,372,036,854,775,807 (inclusive). ($2^{63} - 1$)
- This type is used when a wider range than int is needed.
- Default value is 0L.
- Example : int a = 100000L, int b = -200000L

float:

- Float data type is a single-precision 32-bit IEEE 754 floating point.
- Float is mainly used to save memory in large arrays of floating point numbers.
- Default value is 0.0f.
- Float data type is never used for precise values such as currency.
- Example : float f1 = 234.5f

double:

- double data type is a double-precision 64-bit IEEE 754 floating point.
- This data type is generally used as the default data type for decimal values. generally the default choice.
- Double data type should never be used for precise values such as currency.
- Default value is 0.0d.
- Example : double d1 = 123.4

boolean:

- boolean data type represents one bit of information.
- There are only two possible values : true and false.
- This data type is used for simple flags that track true/false conditions.
- Default value is false.
- Example : boolean one = true

char:

- char data type is a single 16-bit Unicode character.
- Minimum value is '\u0000' (or 0).
- Maximum value is '\uffff' (or 65,535 inclusive).
- Char data type is used to store any character.
- Example . char letterA ='A'

Reference Data Types:

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, Ramesh etc.
- Class objects, and various type of array variables come under reference data type.
- Default value of any reference variable is null.
- A reference variable can be used to refer to any object of the declared type or any compatible type.
- Example : Animal animal = new Animal("giraffe");

Examples:

// write a java program to verify the default values of primitive data types

```
public class CLASSVARIABLES {  
    static int i;  
    float f;  
    char c;  
    byte b;  
    short s;  
    long l;  
    double d;  
    boolean bo;  
    void fun()  
    {
```

```
System.out.println(i);
System.out.println(f);
System.out.println(c);
System.out.println(b);
System.out.println(s);
System.out.println(l);
System.out.println(d);
System.out.println(bo);

}

public static void main(String[] args) {
    CLASSVARIABLES cl = new CLASSVARIABLES();
    cl.fun();
    System.out.println("in side main");
    System.out.println(i);
}
//write a java program to convert one data type values into another
public class CONVERSIONS {

    public static void main(String[] args) {
        int i=20;
        System.out.println(i);
        float f=i;
        System.out.println(f);
        //long l=f; cannot convert from float to long
        long l = i;
        System.out.println(l);
        double d=l;
    }
}
```

```
        System.out.println(d);  
        double d1=i;  
        System.out.println(d1);  
        double d2=f;  
        System.out.println(d2);  
        // short s=i;      cannot convert int to short  
        boolean b=false;  
        System.out.println(b);  
        // int i1=b; cannot convert from boolean to int  
    }  
}
```

Note: Java does not support garbage values. Every local variable must be initialized. If the data members of a class are not initialized, JVM assign the default values for the variables. The default values for Primitive data types are:

byte->0
short->0
int->0
long->0
float->0.0F
double->0.0
char-> " "(blank space)
boolean->false

NOTE : Any integer value assigned to a variable is by default treated as 'int' and the floating-point value as 'double' by Java. That is why, a float value is assigned to a variable of 'float' datatype using the letter 'F' as follows:

```
float pi = 3.1428F;
```

// write a program to initialize the primitive data types

```
public class PRIMITIVETYPES {  
    public static void main(String[] args) {  
        int i=10;  
        float f=20.0f;
```

```
char c='r';
byte b=40;
short s=1000;
long l=200000;
double d=2000.34;
boolean bo=true;
System.out.println(i);
System.out.println(f);
System.out.println(c);
System.out.println(b);
System.out.println(s);
System.out.println(l);
System.out.println(d);
System.out.println(bo);
}

}
```

CHAPTER7

CONTROL STRUCTURES

Control Structures in Java

Control structures are the statements that specify the flow of control in a Java program. The statements used in a program are classified into 3 types. They are:

- 1) Sequential statements.
- 2) Selection statements.
- 3) Iterative statements.

1) Sequential statements: Here, the statements are written in sequential and they are executed in the same order. If the statements are in sequential, there is no ambiguity for the JVM to execute which statement when.

2) Selection statements: These statements are also called 'Branching Statements'. Because they are used to transfer control from one statement to another. These statements include:

a) if statement: 'if' is an example for conditional branching statement because this statement is used with conditions. If the condition is 'true', one set of statements are executed. If it is 'false', the other set are executed. There are 4 forms of 'if statement. They are:

i) Simple-if: Here, we condition only the 'true' part of a condition and ignore the 'false' part. Its syntax is:

```
if(condition)
{
    // statements to be executed when the condition is 'true'.
}
```

-> The following example program demonstrates Simple-if statement:

```
/** Discount Calculation Program Using Simple-if statement */
class SimpleIf
{
    public static void main(String args[])
    {
        float pamt=1999.00F,disc;
        if (pamt >= 2000)
        {
            disc = pamt * 10 / 100;
        }
    }
}
```

```
pamt = pamt - disc;  
}  
  
System.out.println("You Have to Pay Rs." + pamt);  
} // end of main()  
} // end of class
```

ii) if-else statement: Here, both the 'true' and 'false' parts of a condition are considered. Its syntax is :

```
if(condition)  
{  
    // statements to be executed when the condition is 'true'.  
}  
else  
{  
    // statements to be executed when the condition is 'false'.  
}
```

-> The following example program demonstrates if-else statement :

```
// Program for if-else  
class IfElse  
{  
    public static void main(String args[])  
    {  
        int n=25;  
        if(n%2==0)  
            System.out.println(n + " is an even number");  
        else  
            System.out.println(n + " is an odd number");  
    } //main  
} //class
```

CALL:9010990285/7893636382

iii) Nested-if: Here, a condition is defined inside another condition. The inner condition may raise either in the 'true' part or 'false' part of both of main condition. Its syntax is :

```
if(condition)
{
    [if(condition) {} else {} ]
}
else
{
    [if(condition) {} else {} ]
}
```

->The following program demonstrates nested-if statement

```
/** Program to generate Electricity Bill Using Nested-If statement */
```

```
class NestedIf
{
    public static void main(String args[])
    {
        int pres = 2000,prev =1950,units,uprice, billamt;
        units = pres - prev;
        if(units <= 100)
            uprice = 2;
        else
        {
            if(units > 100 && units < 500)
                uprice = 4;
            else
                uprice = 5;
        }
        billamt = units * uprice;
    }
}
```

CALL:9010990285/7893636382

```
System.out.println("You Have to Pay Rs."+ billamt);
} //main()
} //class
```

IV: Else-if ladder: Here, every time when a condition becomes false, one more condition will raise. Its syntax is :

```
if(condition-1) { }
else if(condition-2) { }
else if( condition-3) { } ...
else if(condition-n) { }
else { }
```

-> In the above syntax, the last else block will execute when all the conditions from condition-1 to condition-n becomes 'false'.

-> The following program demonstrates Else-if ladder :

```
/** Program to Calculate Students Grade Using Else-If Ladder */
class ElseIfLadder
{
    public static void main(String args[])
    {
        int m1=70,m2=80,m3=90,total,avg;
        total = m1 + m2 + m3;
        avg = total / 3;
        if(m1<35 || m2<35 || m3<35)
            System.out.println("Sorry! You are Failed");
        else if(avg >= 75)
            System.out.println("Passed In Distinction");
        else if(avg>=60 && avg<75)
            System.out.println("Passed in First Class");
        else if(avg>=50 && avg<60)
```

```
System.out.println("Passed in Second Class");
else
    System.out.println("Passed in Third Class");
} //main()
} //class
```

b) Switch statement:

-> switch is an example for selection statement where we are given a list of options and asked to select one among them. Based on the option selected, the corresponding statements are selected. Its syntax is :

```
switch(expression)
{
    case value-1 : statements;
                    break;
    case value-2 : statements;
                    break;
    .....
    case value-n : statements;
                    break;
    default : default statements;
}
```

The following rules apply to a switch statement:

- The variable used in a switch statement can only be a byte, short, int, or char.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The value for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.

- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

->**The following program demonstrates switch statement :**

```
/** Switch Demo-1 */

class SwitchDemo

{

public static void main(String args[])

{

char ch = 'w';

switch(ch)

{

case 'a' :

case 'e' :

case 'i' :

case 'o' :

case 'u' : System.out.println("It is a Vowel");

break;

default : System.out.println("It is not a Vowel");

}

}//end of main()

}// end of class
```

3) Iterative Statements:

->Iteration means repeated execution of statements. In most situations, we need to execute a set of statements repeatedly for a given number of times.

->Writing the same set of statements repeatedly will have the following drawbacks.

1) Wastage of time.

2) Wastage of memory.

3) Chances of creating errors.

-> To overcome this problem, we go to the concept of loops. Using loops, we will write the statements only once, and execute them as many no. of times as required.

->To implement loops, we need 3 statements. They are:

- 1) Initialization (initializing counter var)
- 2) Increment/decrement (counter value)
- 3) Condition.

->If the condition is 'true', the loop is reentered. If it is 'false', the loop is terminated. Java supports the following loops.

1) do-while loop :

The syntax of do-while loop is :

```
-----
initialization;
do
{
---
inc/dec;
---
}while(condition);
```

-> The following program demonstrates do-while loop

```
//Program for do-while loop
//This program prints sum of first n numbers
class DoWhileDemo
{
    public static void main(String args[])
    {
        int n=10,sum=0;
        int i=0;//initialisation
        do
```

```
{  
    i++; //increment  
    sum=sum+i;  
}while(i<n); //condition  
System.out.println("Sum of first " + n + " numbers is " + sum);  
}//main  
}//class
```

NOTE:

- > Because the condition is placed at the end of the loop, do-while is also called "exit-restricted loop".
- > In do-while loop, irrespective of the value of the condition, the statements are executed for at least one time.

2) while loop :

The syntax of do-while loop is :

```
----  
initialisation;  
while(condition)  
{  
---  
inc/dec;  
---  
}
```

->The following program demonstrates while loop.

```
//Program for while loop  
//This program prints sum of first n numbers  
class WhileDemo  
{  
    public static void main(String args[])  
    {
```

```

int n=10,sum=0;
int i=0;//initialisation
while(i<=n) //condition
{
    sum=sum+i;
    i++; //increment
}
System.out.println("Sum of first " + n + " numbers is " + sum);
}//main
}//class

```

NOTE :

- > Because the condition is placed at the entry of the loop, do-while is also called “entry-restricted loop”.
- > In while loop, without executing the statements for atleast one time, we can terminate the loop.

3) for loop :

->for is the most repeatedly used and simple loop in programming .Its syntax is :

```

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

```

Here is the flow of control in a for loop:

1. The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
2. Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement past the for loop.
3. After the body of the for loop executes, the flow of control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the Boolean expression.
4. The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step,then Boolean expression). After the Boolean expression is false, the for loop terminates.

->The following program demonstrates for loop.

```
/** Program to print factorial of a given number using 'for loop'. */

class ForDemo

{

public static void main(String args[])

{

int n=5,i,fact=1;

for(i=1;i<=n;i++)

{

fact=fact*i;

}

System.out.println("The factorial of " + n + " is " + fact);

}//main

}//class
```

NOTE:

-> Because the condition is placed at the entry of the loop, for is also called “entry-restricted loop”.

-> In for loop also, without executing the statements for atleast one time, we can terminate the loop.

Practice examples:

// write a program to find factorial of a given number

```
package LOOPS.CORE;

import java.util.Scanner;

public class Factorial {

    static int Factorial(int n){

        int fact=1;

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

```
fact=fact*i;  
return fact;  
}  
  
public static void main(String[] args) {  
    Scanner sc = new Scanner(System.in);  
    System.out.println("Enter The First Number:");  
    int n = sc.nextInt();  
    int f = Factorial(n);  
    System.out.println("Factorial of given number is: " +f);  
}  
}
```

// write a program to find sum of digits

```
package LOOPS.CORE;  
import java.util.Scanner;  
public class RevereseNumber {  
    static void ReverseNumber(int n){  
        int rn,sd,nd;  
        rn=sd=nd=0;  
        while(n>0)  
        {  
            rn=rn*10+n%10;  
            sd=sd+n%10;  
            nd=nd+1;  
            n=n/10;  
        }
```

```
System.out.println("Revers of The Given Number: "+rn);
```

```
System.out.println("Sum Of Digits: "+sd);
System.out.println("Number Of Digits:" +nd);
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter The First Number:");
    int n = sc.nextInt();
    ReverseNumber(n);

}

// print the table

package LOOPS.CORE;
import java.util.Scanner;
public class Tables {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter The Number:");
        int a = sc.nextInt();
        for (int i = 1; i <=10; i++) {
            int product = a*i;
            System.out.println(a+ " X " +i +" = "+product);
        }
    }
}

// nested loops

package LOOPS.CORE;
```

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

Enhanced for loop in Java:

Compared to other changes in Java 5, this is a smaller change. The idea behind this change is to improve how we access collection elements and arrays.

Syntax:

The syntax of enhanced for loop is:

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

{}

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

Example:

```
Public class Test {  
    public static void main(String args[]){  
        int [] numbers = {10, 20, 30, 40, 50};  
  
        for(int x : numbers ){  
            System.out.print( x );  
            System.out.print(",");  
        }  
        System.out.print("\n");  
        String [] names = {"James", "Larry", "Tom", "Lacy"};  
        for( String name : names ) {  
            System.out.print( name );  
            System.out.print(",");  
        }  
    }  
}
```

Advantages of for-each Loop:

- Less error prone code
- Improved readability
- Less number of variables to clean up

Limitations:

- Cannot be used where the collection is to be altered while traversing/looping through

- My not be suitable while looping through multiple collections in parallel

The break Keyword:

The *break* keyword is used to stop the entire loop. The *break* keyword must be used inside any loop or a switch statement.

The *break* keyword will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

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

```
break;
```

Example:

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

The continue Keyword:

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 flow of control to immediately jump to the update statement.

- In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression.

Syntax:

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

```
continue;

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

        for(int x : numbers ){
            if( x == 30 ){
                continue;
            }
            System.out.print( x );
            System.out.print("\n");
        }
    }
}
```

More practice programs:

```
// print odd no's up to 50

class Odd

{
    public static void main(String args[])
    {
        int n=1;

        System.out.println("odd no's are:");

        while(n<100)
        {
            System.out.print("\t"+n);

            n+=2;
        }
    }
}
```

```
}
```

```
D:\prr\Core java>javac Odd.java
```

```
D:\prr\Core java>java Odd
```

```
odd no's are:
```

1	3	5	7	9	11	13	15	17	
19	21	23	25	27	29	31	33	35	37
39	41	43	45	47	49	51	53	55	57
59	61	63	65	67	69	71	73	75	77
79	81	83	85	87	89	91	93	95	97
99									

```
// Test whether a number is even or odd
```

```
class EvenOROdd
```

```
{
```

```
    public static void main(String[] args)
    {
        int a=20;
        if(a%2==0)
            System.out.println("It is even number");
        else
            System.out.println("It is odd number");
    }
}
```

```
D:\prr\Core java>javac EvenOROdd.java
```

```
D:\prr\Core java>java EvenOROdd
```

It is even number

// given no is prime or not

```
class PrimeOrNot
```

```
{
```

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

```
{
```

```
        int i,j,k,l;
```

```
        i=9;
```

```
        j=2;
```

```
        l=1;
```

```
        while(j<9)
```

```
{
```

```
            k=i%j;
```

```
            if(k==0)
```

```
                l=k;
```

```
                j++;
```

```
}
```

```
            if(l==0)
```

```
{
```

```
                System.out.println("not prime");
```

```
}
```

```
        else
```

```
{
```

```
                System.out.println("prime");
```

```
}
```

```
}
```

```
}
```

```
D:\prr\Core java>javac PrimeOrNot.java
```

```
D:\prr\Core java>java PrimeOrNot
```

```
not prime
```

//display a multiplication table

```
class MultificationTable
```

```
{
```

```
    public static void main(String args[])
    {
        int i=2,j;
```

```
        System.out.println("Multiplication table is=:");
        for(j=1;j<=10;j++)
        {
            System.out.println(i+"*"+j+"="++(i*j));
        }
    }
}
```

```
D:\prr\Core java>javac MultificationTable.java
```

```
D:\prr\Core java>java MultificationTable
```

```
Multiplication table is=:
```

```
2*1=2
```

```
2*2=4
```

```
2*3=6
```

```
2*4=8
```

2*5=10

2*6=12

2*7=14

2*8=16

2*9=18

2*10=20

/* display the stars the following

```
*  
* *  
* * *  
* * * *  
* * * * *  
*/  
  
class StarsTest  
{  
    public static void main(String args[])  
    {  
        int stars=1,lines=5,spaces=5;  
  
        int i=1,j,k;  
  
        for(i=0;i<lines;i++)  
        {  
            // print spaces first  
  
            for(k=0;k<spaces;k++)
```

```
        System.out.print(" ");

        for(j=0;j<(stars);j++)

            System.out.print("* ");

        spaces ;

        stars +=1;

        System.out.println( );

    }

}

}
```

D:\prr\Core java>javac StarsTest.java

D:\prr\Core java>java StarsTest

```
*  
* *  
* * *  
* * * *  
* * * * *
```

// print the Fibonacci series up to 10

```
class Fibonacci

{

    public static void main(String args[])

    {

        int i=0;

        int j=1,c;

        System.out.print(i);

        System.out.print("\t"+j);
```

```
for(int k=3;k<=10;k++)  
{  
    c=i+j;  
    System.out.print("\t"+c);  
    i=j;  
    j=c;  
}  
}  
}
```

D:\prr\Core java>javac Fibonacci.java

D:\prr\Core java>java Fibonacci

0 1 1 2 3 5 8 13 21 34

// print prime no's

```
class PrimeSeries  
{  
    public static void main(String args[])  
    {  
        int k,m,j;  
        for(j=2;j<100;j++)  
        {  
            k=0;  
            for(int i=1;i<=j;i++)  
            {  
                m=j%i;  
                if(m==0)
```

```
        k=k+1;

    }

    if(k==2)

        System.out.print("\t"+j);

    }

}

}
```

D:\prr\Core java>javac PrimeSeries.java

D:\prr\Core java>java PrimeSeries

2	3	5	7	11	13	17	19	23	29	31	37	41	43
47	53	59	61	67	71	73	79	83	89	97			

//Test whether a character is vowel or not

```
import java.io.*;

class ChTest

{

    public static void main(String args[])throws IOException

    {

        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

        char ch;

        System.out.print("Enter any character: ");

        ch=(char)br.read( );

        if(ch=='a'||ch=='A'||ch=='e'||ch=='E'||ch=='i'||ch=='I'||ch=='o'||ch=='O'||c
h=='u'||ch=='U')

            System.out.println("Vowel");

        else

            System.out.println("Constant");
```

```
}
```

```
}
```

```
D:\prr\Core java>javac ChTest.java
```

```
D:\prr\Core java>java ChTest
```

```
Enter any character: s
```

```
Constant
```

```
D:\prr\Core java>java ChTest
```

```
Enter any character: o
```

```
Vowel
```

// test whether a number is perfect square or not

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

```
class PerfectSquare1
```

```
{
```

```
    public static void main(String args[])throws IOException
```

```
    {
```

```
        BufferedReader br=new BufferedReader(new
```

```
                                         InputStreamReader(System.in));
```

```
        int i=0,x=0;
```

```
        System.out.print("enter any number: ");
```

```
        i=Integer.parseInt(br.readLine( ));
```

```
        x=(int)Math.sqrt(i);
```

```
        if(i==(x*x))
```

```
            System.out.println("Perfect square ");
```

```
        else
```

```
            System.out.println("Not a perfect Square ");
```

```
    }  
}  
  
}
```

D:\prr\Core java>javac PerfectSquare1.java

D:\prr\Core java>java PerfectSquare1

Enter any number: 5

Not a perfect Square

D:\prr\Core java>java PerfectSquare1

Enter any number: 9

Perfect square

// given point lies inside the circle or out side the circle or on the circle

```
import java.io.*;  
  
class PointLies  
{  
  
    public static void main(String args[])throws IOException  
{  
        BufferedReader br=new BufferedReader(new  
                                         InputStreamReader(System.in));  
  
        int x=0,y=0,r=0;  
  
        System.out.println("Enter x & y co ordinates : ");  
  
        x=Integer.parseInt(br.readLine());  
  
        y=Integer.parseInt(br.readLine());  
  
        System.out.println("Enter Radius :");  
  
        r=Integer.parseInt(br.readLine());  
  
        int a=(x*x)+(y*y);  
  
    }
```

```
int b=(r*r);

if(a==b)

    System.out.println("Point lies on the Circle");

else if(a>b)

    System.out.println("Point lies out side the Circle");

else

    System.out.println("Point lies in side the Circle");

}

}
```

D:\prr\Core java>javac PointLies.java

D:\prr\Core java>java PointLies

Enter x & y co ordinates :

10

20

Enter Radius :

4

Point lies out side the Circle

// to find the Factorial value

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

```
class Factorial
```

```
{
```

```
    public static void main(String args[])throws IOException
```

```
{
```

```
    BufferedReader br=new BufferedReader(new
```

```
        InputStreamReader(System.in));
```

CALL:9010990285/7893636382

```
int fact=1;

System.out.print("Enter any Number : ");

int n=Integer.parseInt(br.readLine( ));

if(n==0)

System.out.println("Factorial of 0 is 1");

while(n>0)

{

    fact=fact*n;

    n =1;

}

System.out.println("Factorial =" +fact);

}

}
```

D:\prr\Core java>javac Factorial.java

D:\prr\Core java>java Factorial

Enter any Number : 5

Factorial =120

// sum of the digits

```
import java.io.*;

class SumDigits

{

    public static void main(String args[])throws IOException

    {

        BufferedReader br=new BufferedReader(new
```

```
InputStreamReader(System.in));  
  
int n=0,sum=0,d=0;  
  
System.out.print("Enter any Number : ");  
  
n=Integer.parseInt(br.readLine( ));  
  
do  
  
{  
  
    d=n%10;  
  
    sum+=d;  
  
    n=n/10;  
  
}  
  
while(n>0);  
  
System.out.println("Sum of digits =" +sum);  
  
}  
  
}  
  
D:\prr\Core java>java SumDigits  
  
Enter any Number : 236  
  
Sum of digits =11  
  
/* all the 4 number digits Square of the 1st 2 digits are perfect squire & also last 2 digits */  
  
import java.io.*;  
  
class PerfectSquare  
  
{  
  
    public static void main(String args[])  
  
    {  
  
        int nl=0,nr=0,temp=0,x=0,y=0,z=0;  
  
    }
```

```
for(int i=1000;i<=9999;i++)  
{  
    temp=i;  
    x=(int)Math.sqrt(temp);  
    {  
        if(temp==x*x)  
        {  
            nl=temp/100;  
            nr=temp%100;  
            y=(int)Math.sqrt(nl);  
            z=(int)Math.sqrt(nr);  
            if((nl==y*y)&&(nr==z*z))  
                System.out.print(temp+"\t");  
        }  
    }  
}  
}
```

D:\prr\Core java>javac PerfectSquare.java

D:\prr\Core java>java PerfectSquare

1600 1681 2500 3600 4900 6400 8100

CHAPTER8

ARRAYS

AND

STRINGS

Arrays in Java

CALL:9010990285/7893636382

-> An array is defined as a collection of values of similar data type. It is a group of contiguous or related data items that share a common name. Java language supports two types of arrays namely: Single-dimensional (1-D) and Multi-dimensional (2-D) arrays.

One-Dimensional arrays:

-> In these types of arrays, in each location of the array, we can store only one value. The syntax for declaring an array is:

Type array-name [];

-> Once the array is declared, we need to create it in the memory. Java allows us to create arrays using 'new' operator as shown below.

Array-name = new type [size];

For example, consider the following statements.

```
int a[];
```

a = new int[10]; Here, we are reserving memory for variable 'a' to store 10 integer values(40 bytes). Once an array is declared, memory is created sequentially for the variable.

-> Instead of declaring arrays using the above syntax, we can also declare them as per the following syntax:

```
type array-name[] = new type[size];
```

E.g.: int a [] = new int[10];

-> If we want to initialize array variables, we can use the following syntax:

```
type array-name[] = {list-of-values};
```

Eg: int a[] = {10,20,30,40,50};

-> The following program demonstrates the concept of 1-D arrays:

```
/* Program to sort a given list of array values */
```

```
class Sorting
```

```
{
```

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

```
{
```

```
    int a[] = {96,34,78,12,45};
```

```
    int n = a.length;
```

```
    System.out.print("Elements Before Sorting : ");
```

```
for(int i=0;i<n;i++)  
{  
    System.out.print(" " + a[i]);  
}  
  
System.out.println(" ");  
/* Logic for sorting */  
for(int i=0;i<n;i++)  
{  
    for(int j=i+1;j<n;j++)  
    {  
        if(a[i] > a[j])  
        {  
            int temp = a[i];  
            a[i] = a[j];  
            a[j] = temp;  
        }  
    }  
}  
  
System.out.print("Elements After Sorting : ");  
for(int i=0;i<n;i++)  
{  
    System.out.print(" " + a[i]);  
}  
} // end of main  
} // end of class
```

Two-Dimensional arrays (2-D)

-> The arrays that we discussed above are called one-dimensional arrays because in each location of the array, we can store exactly one value. But, in some cases, under each position

of the array, we need to store multiple values. Such arrays are called two-dimensional or 2-D arrays. That is, every position of the array is again an array.

->Tables and matrices are the best examples for 2-D arrays. The syntax for creating a 2-D array is as follows :

```
type array-name[][];
```

```
array-name = new type[row][column];
```

For example, int a[][];

a = new int[5][5]; That is, 'a' is a two-dimensional array variable that can store a max. of 5 row values. In each row, we can store a max. of 5 column values. Totally under 'a', we can store $5 * 5 = 25$ values.

This would produce following result:

```
hello
```

Note: The String class is immutable; so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters then you should use String Buffer & String Builder Classes.

// find sum of 2 matrices

```
import java.io.*;  
  
class Arr  
{  
  
    // instance variables  
    int r,c;  
    int arr[][];  
  
    // constructor  
    Arr(int r,int c)  
    {  
        this.r=r;  
        this.c=c;  
        arr=new int[r][c];  
    }  
}
```

```
// to receive matrix elements from key board

int [][] getArray( ) throws IOException

{

    BufferedReader br=new BufferedReader(new

                                InputStreamReader(System.in));

    for(int i=0;i<r;i++)

    {

        for(int j=0;j<c;j++)

        {

            System.out.print("Enter element : ");

            arr[i][j]=Integer.parseInt(br.readLine( ));

        }

    }

    return arr;

}

// to find sum of 2 matrices

int [][] findSum(int a[][],int b[][])

{

    int temp[][]=new int[r][c];

    for(int i=0;i<r;i++)

    {

        for(int j=0;j<c;j++)

        {

            temp[i][j]=a[i][j]+b[i][j];

        }

    }

}
```

```
        }

        return temp;
    }

    // display the sum matrix

    void putArray(int res[][][])
    {
        for(int i=0;i<r;i++)
        {
            for(int j=0;j<c;j++)
            {
                System.out.print(res[i][j]+\t");
            }
            System.out.println("\n");
        }
    }
}

class MatrixSum
{
    public static void main(String[] args) throws IOException
    {
        // create 2 objects to arr

        Arr obj1=new Arr(3,3);

        Arr obj2=new Arr(3,3);

        // take 3 array references

        int x[][],y[][],z[][];
```

```
System.out.println("enter matrix 1 :");

x=obj1.getArray( );

System.out.println("enter matrix 2 :");

y=obj2.getArray( );

z=obj1.findSum(x,y);

System.out.println("\n The sum matrix is : ");

obj2.putArray(z);

}

}
```

D:\prr\Core java>javac MatrixSum.java

D:\prr\Core java>java MatrixSum

enter matrix 1 :

Enter element : 1

Enter element : 2

Enter element : 3

Enter element : 4

Enter element : 5

Enter element : 6

Enter element : 7

Enter element : 8

Enter element : 9

enter matrix 2 :

Enter element : 1

Enter element : 2

CALL:9010990285/7893636382

Enter element : 3

Enter element : 4

Enter element : 5

Enter element : 6

Enter element : 7

Enter element : 8

Enter element : 9

The sum matrix is :

2	4	6
8	10	12
14	16	18

// find product of 2 matrices

```
import java.io.*;  
  
class Arr1  
{  
    // instance variables  
    int r1,c1;  
    int arr1[][];  
    // constructor  
    Arr1(int r1,int c1)  
    {  
        this.r1=r1;  
        this.c1=c1;  
        arr1=new int[r1][c1];  
    }  
}
```

```
// to receive matrix elements from key board

int [][] getArray( ) throws IOException

{

    BufferedReader br=new BufferedReader(new

                                InputStreamReader(System.in));

    for(int i=0;i<r1;i++)

    {

        for(int j=0;j<c1;j++)

        {

            System.out.print("Enter element :");

            arr1[i][j]=Integer.parseInt(br.readLine( ));

        }

    }

    return arr1;

}

class Arr2

{

    // instance variables

    int r2,c2;

    int arr2[][];

    // constructor

    Arr2(int r2,int c2)

    {



}
```

```
        this.r2=r2;

        this.c2=c2;

        arr2=new int[r2][c2];

    }

// to receive matrix elements from key board

int [][] getArray( ) throws IOException

{

BufferedReader br=new BufferedReader(new

                        InputStreamReader(System.in));

for(int i=0;i<r2;i++)

{

    for(int j=0;j<c2;j++)

    {

        System.out.println("Enter element :");

        arr2[i][j]=Integer.parseInt(br.readLine( ));

    }

}

return arr2;

}

class Prod

{

    int r1,c2,c1;

    Prod(int r1,int c2,int c1)

    {

        CALL:9010990285/7893636382
    }
}
```

```
this.r1=r1;  
this.c2=c2;  
this.c1=c1;  
}  
  
// to find Prod of 2 matrices  
  
int [][] findProd(int a[][][],int b[][][]){  
  
    int temp[][]=new int[r1][c2];  
  
    for(int i=0;i<r1;i++)  
  
    {  
  
        for(int j=0;j<c2;j++)  
  
        {  
  
            for(int k=0;k<c1;k++)  
  
            {  
  
                temp[i][j]+=a[i][k]*b[k][j];  
            }  
        }  
    }  
    return temp;  
}  
  
// display the Prod matrix  
  
void putArray(int res[][][]){  
  
    for(int i=0;i<r1;i++)  
  
    {  
        for(int j=0;j<c2;j++)  
    }
```

```
for(int j=0;j<c2;j++)  
{  
    System.out.print(res[i][j]+\t");  
}  
System.out.println("\n");  
}  
}  
}  
  
class MatrixProd  
{  
    public static void main(String[] args) throws IOException  
    {  
        BufferedReader br=new BufferedReader(new  
                InputStreamReader(System.in));  
        System.out.println("Enter first matrix r1,c1");  
        int r1=Integer.parseInt(br.readLine());  
        int c1=Integer.parseInt(br.readLine());  
        System.out.println("Enter second matrix r2,c2");  
        int r2=Integer.parseInt(br.readLine());  
        int c2=Integer.parseInt(br.readLine());  
        if(c1==r2)  
        {  
            // create 3 objects to arr  
            Arr1 obj1=new Arr1(r1,c1);  
            Arr2 obj2=new Arr2(r2,c2);  
        }  
    }  
}
```

```
Prod p=new Prod(r1,c2,c1);

// take 3 array references

int x[][][],y[][][],z[][][];

System.out.println("\n enter matrix 1 :");

x=obj1.getArray( );

System.out.println("\n enter matrix 2 :");

y=obj2.getArray( );

z=p.findProd(x,y);

System.out.println("\n The prod matrix is :");

p.putArray(z);

}

else

System.out.println("can not product of matrix");

}

}

D:\prr\Core java>javac MatrixProd.java

D:\prr\Core java>java MatrixProd
```

Enter first matrix r1,c1

3

2

Enter second matrix r2,c2

2

3

enter matrix 1 :

Enter element : 1

CALL:9010990285/7893636382

Enter element : 2

Enter element : 3

Enter element : 4

Enter element : 5

Enter element : 6

enter matrix 2 :

Enter element : 1

Enter element : 2

Enter element : 3

Enter element : 4

Enter element : 5

Enter element : 6

The prod matrix is :

9 12 15

19 26 33

29 40 51

STRINGS

Strings represent a group of characters. In C or C++ strings are character arrays. \ 0 is null character. It is end of string.

In java strings are not character arrays. Strings are string class object in object. A string is an object of string class. Any string is an object of string class in java.

Creating strings:

- 1) We can declare a string type variable & store directly in to it.

```
String str="hello";
```

- 2) We can create a string class object using new operator and pass the string to it.

```
String s1 =new String ("Hai");
```

- 3) We can create a string by converting a character array in to a string object.

```
Char arr[ ] = {'a','b','c','d','e','f'};
```

```
String s2 = new String (arr);
```

```
String s2=new String (arr,1,3);
```

String methods:

Java.lang.string:

- 1) String concat (String str): Concat means method. A method can form a task .Concatenates the calling string with str.

Note: + will also do the same

Concat is a method it joins two strings. When you call concat pass a string object to it.

Ex: String s1 = "Hydera";

```
String s2 = "bad";
```

```
String x = s1.concat(s2);
```

Display Hyderabad.

+ is called concatenation operator.

Concat: joining string at the end another string.

- 2) int length (): returns the length of a string.

Ex: String s1 = "Hydera";

```
Int n = s1.length();
```

- 3) char charAt(int index):

Returns the character at the specified location (from 0).

- 4) int compareTo (string Str): Returns a negative value, if the calling String comes before str in dictionary order, a +ve value, if the String comes after str, or 0, if the String are equal.

EX:

```
int n=s1.compareTo(s2);
```

```
if s1==s2,    n=0
```

```
if s1>s2,    n>0
```

```
if s1<s2,    n<0
```

```
s1="boy", s2="box";
```

It is a +ve number. X is first, first is less value.

5) boolean equals(String str): Returns true if the calling string equals str. Equals is case sensitive method.

6) boolean equalsIgnoreCase(String str): Same as above. This is case insensitive method.

7) boolean startsWith(String prefix): Returns true if the calling string starts with prefix. Prefix means the string starting with beginning.

8) boolean endsWith(String suffix): Returns true if the invoking string ends with suffix. Suffix means the string ending with beginning.

Note: Above 2 methods use case sensitive comparison.

9) int indexOf(String str): Returns the first occurrence of str in the string.

10) int lastIndexOf(String str): Returns the last occurrence of str in the string.

Note: Both the above methods return -ve value, if str not found in the calling string. Counting starts from 0.

Ex: String str="This is a book";

2 5

```
int n=str.indexOf("is"); // It gives 2 only because first position.
```

```
int n=str.lastIndexOf("is"); // it gives 5 only.
```

11) String replace(char oldChar,char newChar): Returns a new string. That is obtained by replacing characters old char in the string with new char.

12) String substring(int beginIndex): Return a new string consisting of all characters from begin index until the end of the string.

13) String substring(int beginIndex,int endIndex): Returns a new string consisting of all characters from begin index until end index(exclusive). Last character is exclusive.

14) String toLowerCase(): Converts all characters into lower case.

15) String toUpperCase(): Converts all characters into upper case.

16) String trim(): Eliminates all leading & trailing spaces.

Trim means deleting or cutting. Trim method doesn't remove in the middle spaces. It's only remove first spaces & last spaces. Last space is removing, it is trail method.

// understanding the strings

```
class StringsTest

{
    public static void main(String[] args)
    {
        // create 3 strings

        String s1="This is java";

        String s2=new String(" I like it");

        char ch[]={ 'P','o','t','h','u','r','a','i'};

        String s3= new String(ch);

        // display the strings

        System.out.println("S1 = "+s1);

        System.out.println("S2 = "+s2);

        System.out.println("S3 = "+s3);

        // find no . of characters in s1

        System.out.println("length of S1 = "+s1.length( ));

        // join 2 strings

        System.out.println("S1 joined with s2 = "+s1.concat(s2));

        // join 3 strings

        System.out.println(s1+" at "+s3);

        // check the starting of s1

        boolean x=s1.startsWith ("This");

        if(x==true)
```

```
System.out.println("s1 starts with This ");

else

    System.out.println("S1 does not start with This ");

/* extract substring from s1 & s3

String p=s2.subString(0,6);

String q=s3.subString(0);

System.out.println(p+q); */

//change the case of s3

System.out.println("Upper case of s3 = "+s3.toUpperCase( ));

System.out.println("Lower case of S3 = "+s3.toLowerCase( ));

}

}
```

D:\prr\Core java>javac StringsTest.java

D:\prr\Core java>java StringsTest

S1 = This is java

S2 = I like it

S3 = Pothurai

length of S1 = 12

S1 joined with s2 = This is java I like it

This is java at Pothurai

s1 starts with This

Upper case of s3 = POTHURAI

Lower case of S3 = pothurai

// equality of strings

class StringEqual

CALL:9010990285/7893636382

```

{
    public static void main(String args[])
    {
        String a="Hello";
        String b=new String("Hello");
        if(a==b)
            System.out.println("same");
        else
            System.out.println("Not same");
    }
}

```

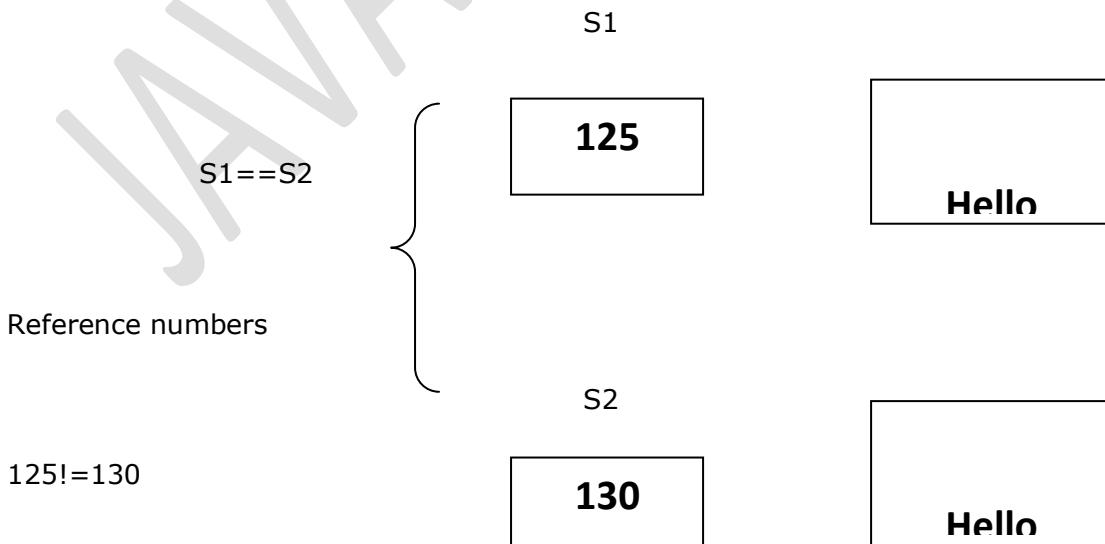
D:\prr\Core java>javac StringEqual.java

D:\prr\Core java>java StringEqual

Not same

Q) What is hash code?

- A) It is a unique identification number that is allotted by JVM to the objects. This number is also called reference number.



So the output is not same. == is only compare reference numbers

CALL:9010990285/7893636382

In the above program correct format is if(s1.equals(s2))

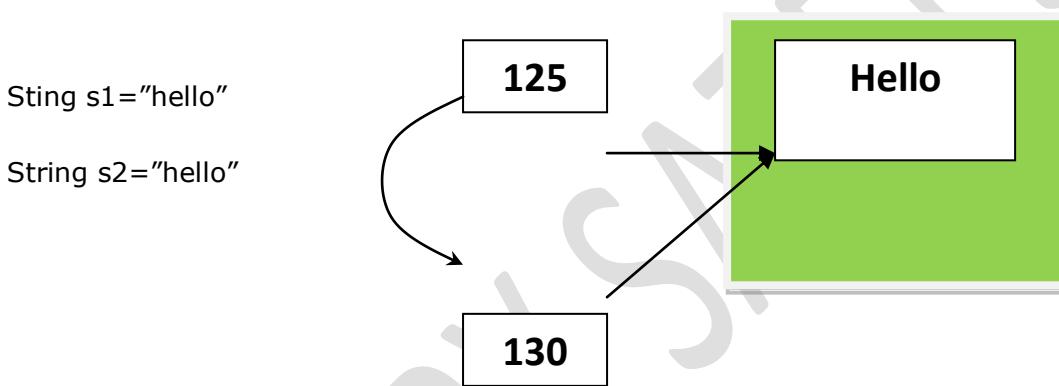
Q) What is the difference between == to and equals method while comparing the strings?

A) == operator compares only the reference of string objects.

Equals method compares the contents of the string objects. Hence the equals' method gives reliable results.

Q) What is string constant pool?

A) It is separate block of memory where the string objects are stored.



125 is copy to another

Same reference no's so

125==125

String constant pool

If(s1==s2)

o/p: equal, because JVM does not waste memory.

Q) What is the difference between the following statements?

a) String s="Hello";

b) String s=new String("Hello");

A) When the first statement is executed JVM searches for the same object in the string constant pool. If the same object is found there then JVM will create another reference to the same object, if the same object not found JVM creates another object & stores it into string constant pool.

When the second statement is executed JVM always create a new object without searching in the string constant pool.

Types of objects:

- 1) Mutable
- 2) Immutable.

1) Mutable:

Mutable objects are the objects whose contents can be modified.

2) Immutable:

Immutable objects are those objects whose contents can not be modified.

// String class object are immutable

```
class Immutable

{
    public static void main(String[] args)
    {
        String s1="hello";
        String s2="hai";
        s1=s1+s2;
        System.out.println(s1);
    }
}
```

D:\prr\Core java>javac Immutable.java

D:\prr\Core java>java Immutable

Hellohai

String buffer: String buffer is a mutable, where as string is immutable.

Creating a string buffer:

- 1) StringBuffer sb=new StringBuffer("Hello");
- 2) StringBuffer sb=new StringBuffer

java.lang. StringBuffer:

- 1) StringBuffer append(x): x may be int, float, double, char, string or StringBuffer. It will be append to the calling StringBuffer.
- 2) StringBuffer insert(int offset,x): x may be int, float, double, char, string or StringBuffer. It will be inserted into the StringBuffer at offset. Insert method is insert a value.
- 3) StringBuffer delete(int start,int end): removes the characters from start to end.
- 4) StringBuffer reverse(): Reverse the character sequence in the StringBuffer
- 5) String toString(): Converting StringBuffer into a string
- 6) int length(): Returns the length of the StringBuffer

What is the difference between a string & StringBuffer?

A) String objects are immutable. StringBuffer objects are mutable.

The methods which directly manipulate the data or not available in string class. Such methods are available in StringBuffer class.

```
// display the full name

import java.io.*;

class Mutable

{
    public static void main(String[] args) throws IOException
    {
        // to accept data from keyboard
        BufferedReader br=new BufferedReader(new
                                         InputStreamReader(System.in)));
    }
}
```

```
System.out.print("Enter sur name : ");

String sur=br.readLine( );

System.out.print("Enter mid name : ");

String mid=br.readLine( );

System.out.print("Enter last name : ");

String last=br.readLine( );

// create String Buffer object

StringBuffer sb=new StringBuffer( );

// append sur,last to sb

sb.append(sur);

sb.append(last);

// insert mid after sur

int n=sur.length( );

sb.insert(n,mid);

// display full name

System.out.println("Full name = "+sb);

System.out.println("In reverse =" +sb.reverse( ));

}

}
```

D:\prr\Core java>java Mutable

Enter sur name : Sunil

Enter mid name : Kumar

Enter last name : Reddy

Full name = SunilKumarReddy

In reverse =yddeRramuKlinuS

CALL:9010990285/7893636382

CHAPTER8

CONSTRCUTORS

Constructors in Java:

- Constructor is a special method that gets invoked “automatically” at the time of object creation.
- Constructor is normally used for initializing objects with default values unless different values are supplied.
- Constructor has the same name as the class name.
- Constructor cannot return values.
- A class can have more than one constructor as long as they have different signature (i.e., different input arguments syntax).
- If you not write any constructor in class, java compiler automatically creates one constructor called default constructor.

-> Apart from the default constructor, we can create our own constructors also. A constructor can be defined with or without parameters but a constructor should not have any return types. Also, a constructor can have any number of parameters.

Rules for Java Constructor:

1. Constructors can use any access modifier, including private.(A private constructor means only code within the class itself can instantiate an object of that type, so if the **private** constructor class wants to allow an instance of the class to be used, the class must provide a static method or variable that allows access to an instance created from within the class.)
2. The constructor name must match the name of the class.
3. Constructors must not have a return type.
4. It's legal (but stupid) to have a method with the same name as the class, but that doesn't make it a constructor. If you see a return type, it's a method rather than a constructor. In fact, you could have both a method and a constructor with the same name the name of the class in the same class, and that's not a problem for Java. Be careful not to mistake a method for a constructor, be sure to look for a return type.
5. If you don't type a constructor into your class code, a default constructor will be automatically generated by the compiler.
6. The default constructor is ALWAYS a no-argument constructor.
7. Every constructor has, as its first statement, either a call to an overloaded constructor (`this()`) or a call to the super class constructor (`super()`), although remember that this call can be inserted by the compiler.
8. A call to `super ()` can be either a no-arg call or can include arguments passed to the super constructor.

9. A no-argument constructor is not necessarily the default (i.e., compiler-supplied) constructor, although the default constructor is always a no-argument constructor. The default constructor is the one the compiler provides! While the default constructor is always a no-argument constructor, you're free to put in your own no argument constructor.
10. You cannot make a call to an instance method, or access an instance variable, until after the super constructor runs.
11. Only static variables and methods can be accessed as part of the call to super() or this(). (Example: super (Animal.NAME) is OK, because NAME is declared as a static variable.)
12. Abstract classes have constructors, and those constructors are always called when a concrete subclass is instantiated.
13. Interfaces do not have constructors. Interfaces are not part of an object's inheritance tree.
14. The only way a constructor can be invoked is from within another constructor.

/* the following programs illustrate the concept of constructors */

Eg : 1)

```
/* Program to demonstrate default constructor */

class ConstructorDemo1

{

int l;

public static void main(String args[])
{
    ConstructorDemo1 c1 =new ConstructorDemo1 ();
    System.out.println("Area of Square = " + c1.l*c1.l);
}
}// end of class ConstructorDemo1
```

Eg: 2)

/* Program to demonstrate constructor without parameters */

```
class Square

{
int l;
Square() // constructor
{
    System.out.println("Inside constructor without parameters ");
}
```

```
I = 100;  
}  
  
void area()  
{  
    System.out.println("Area of Square = " + I * I);  
}  
}// End of class Square  
  
class ConstructorDemo2  
{  
    public static void main(String args[])  
    {  
        Square s = new Square(); // calls constructor method  
        s.area();  
    } // end of main  
} // end of class ConstructorDemo2
```

Eg : 3)

```
/* Program to demonstrate constructor with parameters */  
  
class Square  
{  
    int I;  
    Square(int x) // constructor  
    {  
        System.out.println("Inside constructor with parameters ");  
        I = x;  
    }  
    void area()  
{  
    System.out.println("Area of Square = " + I * I);  
}
```

```
}

}// End of class Square

class ConstructorDemo3

{

public static void main(String args[])

{

Square s = new Square(100); // calls constructor method

s.area();

} // end of main

} // end of class ConstructorDemo3
```

Eg : 4)

```
/* Program to demonstrate multiple constructors */

class Rectangle

{

int l,b;

Rectangle() //



{

System.out.println("Inside constructor without parameters ");

l = 100;

b = 200;

}

Rectangle(int x) //



{

System.out.println("Inside constructor with one parameter ");

l = x;

b = x;

}

Rectangle(int x,int y)
```

```
{  
    System.out.println("Inside constructor with two parameters ");  
    l = x;  
    b = y;  
}  
  
void area()  
{  
    System.out.println("Area of Square = " + l * l);  
}  
}// End of class Rectangle  
  
class ConstructorDemo4  
{  
    public static void main(String args[])  
    {  
        Rectangle r1 = new Rectangle();  
        Rectangle r2 = new Rectangle(15);  
        Rectangle r3 = new Rectangle(50,150);  
        r1.area();  
        r2.area();  
        r3.area();  
    } // end of main  
} // end of class ConstructorDemo4
```

finalize() method:

-> Unlike constructors, a class can have a special method that can be used before a class is garbage collected.

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 program should provide other means of releasing system resources, etc., used by the object. It must not rely on **finalize()** for normal program operation.

this keyword:

-> Java Language a special keyword 'this' that can be used in two ways.

1. To access the instance variables of a class inside a method of that class.
2. To call a constructor of a class from another constructor of that class.

-> In a Java Program, if we have the same variable name both as local and global (instance variable), then, priority is given for local variables only. If we want to access the instance variables of a class inside a method, we can use the keyword called 'this'.

Definition: 'this' is the keyword that points to the object that is executing the block in which the control is currently under execution.

The following program demonstrates the use of 'this' keyword:

```
class ThisDemo1
{
    int a=10,b=20;
    void fun1()
    {
        System.out.println("Inside fun1");
        int a=25;
```

```
this.a=a + 100;  
a=this.a - 25;  
  
System.out.println("local a= " + a);  
  
System.out.println("Instance b= " + b);  
  
System.out.println("Instance var a= " + this.a );  
}  
  
public static void main(String args[]){  
}  
  
ThisDemo2 t1= new ThisDemo2();  
t1.fun1();  
}  
}
```

->The second purpose of 'this' keyword is to call the constructor of a class from another constructor. This calling statement must be the first statement in the calling constructor. The following program illustrates this concept.

/ Program to call a constructor of a class from another constructor */**

```
class ThisDemo3  
{  
    int a,b;  
    ThisDemo3()  
    {  
        System.out.println("Inside constructor without parameters");  
        a=10;  
        b=20;  
    }  
    ThisDemo3(int x)  
    {  
        a=x;  
        b=x;  
    }  
}
```

```
System.out.println("Inside constructor with one parameters");  
}  
  
ThisDemo3(int x,int y)  
{  
    this(x); // x value passed to ThisDemo(int x)  
  
    System.out.println("Inside constructor with two parameters");  
}  
  
public static void main(String args[])  
{  
    ThisDemo3 t1=new ThisDemo3();  
  
    System.out.println("t1.a = " + t1.a);  
  
    System.out.println("t1.b = " + t1.b);  
  
    ThisDemo3 t2=new ThisDemo3(10,20);  
  
    System.out.println("t2.a = " + t2.a);  
  
    System.out.println("t2.b = " + t2.b);  
}  
}
```

NOTE: 1. 'this' keyword must be used only in either constructors or non-static methods of a class. It cannot be used in static methods of a class.

2. Writing two or more this call in the same constructor is wrong.

Example:

```
class Sample{  
    Sample(){  
        ....  
        ....  
    }  
    Sample(int a){  
        this();  
        this();  
        ....  
    }  
}
```

```
....  
}  
}// it is wrong
```

3. Writing the this call as second statement is also wrong

Example:

```
class Ramesh{  
Ramesh(){  
System.out.println("I am in cons");  
this(30);  
}  
System.out.println("some mess");  
}// it is wrong
```

4. Recursive methods calls are allowed in java but recursive constructor calls are not allowed in java.

Static and non-static blocks:

These are a set of statements that can be defined with the keyword static or without static. Any block that starts with static keyword is called a static block. Its syntax is:

```
static
```

```
{  
---  
}
```

->Any block defined without a name or static keyword is called a non-static block. Its syntax is :

```
{  
----  
}
```

IMPORTANT POINTS TO REMEMBER:

1. If you write non static data member inside non static method then it is converted as this.non static data member

2. Whenever we declared a local variable with same name as non static data member then non static data member usage cannot be converted as this.non static data.

How these blocks are executed?

->Whenever we ask JVM to execute a .class file, it checks whether that file is available or not.If available, before loading main() and other static members into memory, it checks whether the program is having any static blocks.If available, these blocks are loaded and executed.

->If in the main(), we are creating the object of the class,before loading the constructor of the class the JVM checks whether the class is having any non-static blocks.If available, they are loaded and executed.Then, the constructor block will gets executed.

->Stated simply, static blocks are executed before main() and non-static blocks are executed before constructors.

Benefits of using static and non-static blocks:

->In real-time applications, before loading the main module of the software, if we want to provide any user-guide information, we will provide it in static blocks.

->A class may have any no. of constructors. If multiple constructors of the class are having some statements in common, instead of writing them in all constructors, we can define them in non-static blocks.

->The following program demonstrates the use of static and non-static blocks :

```
/* Program to implement static and non-static blocks */

class StaticBlockDemo

{
    StaticBlockDemo()
    {
        System.out.println("Inside constructor");
    }

    static //static block
    {
        System.out.println("Inside static block");
    }

    void display()
    {
        System.out.println("Inside display function");
    }
}
```

```

}

{ //non-static block

System.out.println("Inside non-static block");

}

public static void main(String args[])

{

System.out.println("Inside main");

StaticBlockDemo sd = new StaticBlockDemo();

sd.display();

}// end of main()

}// end of class

```

Q) What are the Differences between Constructor & Method?

Constructor	Method
Use to instance of a class	Grouping java statement
No return type	Void (or) valid return type
Same name as class name	As a name except the class method name, begin with lower case.
"This" refer to another constructor in the same class	Refers to instance of class
"Super" to invoke the super class constructor	Execute an overridden method in the super class
"Inheritance" cannot be inherited	Can be inherited
We can overload but we cannot overridden	Can be inherited
Will automatically invoke when an object is created	Method has called explicitly

Wrapper Classes:

Wrapper class: All wrapper classes are defined in `java.lang` package.

A wrapper class wraps contains a primitive data type in its object.

Character is a wrapper class in to a char data type.

Primitive data type Wrapper class:

char	Character
byte	Byte
int	Integer
float	Float
double	Double
long	Long.

1) Character class: The character class wraps a value of the primitive type 'char' an object. An object of type character contains a single field whose type is char.

Constructors:

1) Character (char ch)

Ex: char ch = 'A'

Character obj=new Character (ch);

Methods:

1) char charValue ()

Returns the char value of the invoking object.

Ex: char x =obj.charValue();

2) static boolean isDigit(char ch)

Returns true if ch is a digit (0 to 9) otherwise returns false.

Ex: char = '9'; boolean x = character.isDigit(ch);

3) static boolean isLetter(char ch).

returns true if ch is a letter (A to Z or a to z)

4) static boolean isUpperCase(char ch).

returns true if ch is an upper case letter (A to Z).

5) static boolean isLowerCase(char ch)

returns true if ch is an Lower case letter (a to z).

6) static boolean isSpaceChar(char ch).

returns true if ch is coming from Space Bar.

7) static boolean isWhitespace(char ch)

returns true if ch is coming from Tab, Enter, Backspace.

8) static char toUpperCase (char ch).

converts ch into upper case.

9) static char toLowerCase (char ch)

Converts ch into lowercase.

**Q) Which of the wrapper classes does not a constructor with string parameter? (or)
Which of the wrapper classes contains only one constructor?**

A) Character

// test a character

```
import java.io.*;

class CharacterTest

{
    public static void main(String[] args) throws IOException
    {
        while(true)
        {
            // to accept data from key board
            BufferedReader br=new BufferedReader(new
                InputStreamReader(System.in));
            System.out.println("Enter a char");
        }
    }
}
```

```
char ch=(char)br.read( );

// test the ch

if(ch=='@')

System.exit(0);

else if(Character.isDigit(ch))

    System.out.println("It is a digit");

else if(Character.isUpperCase(ch))

    System.out.println("It is a capital letter");

else if(Character.isLowerCase(ch))

    System.out.println("It a small Letter");

else if(Character.isSpaceChar(ch))

    System.out.println("It is coming from space bar");

else if(Character.isWhitespace(ch))

    System.out.println("It is a white space");

else

    System.out.println("Sorry, i don't know that character");

}

}

D:\prr\Core java>javac CharacterTest.java

D:\prr\Core java>java CharacterTest
```

Enter a char

p

It a small Letter

Enter a char

CALL:9010990285/7893636382

4

It is a digit

Enter a char

~

Sorry, i don't know that character

Enter a char

@

2) Byte class: The byte class wraps a value of primitive type 'byte' in an object. An object of type byte contains a single field whose type is byte.

Constructor:

- 1) Byte(byte num)
- 2) Byte(String str)

Ex: 1) byte b=99;

```
Byte obj=new Byte(b);
```

```
2) String str="120"
```

```
Byte obj=new Byte(str);
```

Methods:

→1) byte byteValue()

Returns the value of invoking object as a byte.

→2) int compareTo(Byte b)

Compares the numerical value of invoking object with that of 'b'. returns 0, if the values all equal. Returns a -ve value, if the invoking object has a lower value. Returns a +ve value, if the invoking object has a grater value.

Ex: int n=b1.compareTo(b2)

```
If b1==b2;           n==0;  
If b1<b2;           n== -ve;  
If b1>b2;           n== +ve;
```

CALL:9010990285/7893636382

→3) static byte parseBytes(String str) throws NumberFormatException

Returns the byte equivalent of the number contained in the string specified by 'str'.

→4) String toString()

Returns a string that contains the decimal equivalent of the invoking object.

→5) static ByteValueOf(String str) throws NumberFormatException

Returns a byte object that contains the value specified by string 'str'.

// creating & comparing byte objects

```
import java.io.*;

class Bytes

{
    public static void main(String args[])throws IOException
    {
        BufferedReader br=new BufferedReader(new
                                         InputStreamReader(System.in));

        System.out.print("enter a byte no : ");

        String str=br.readLine( );

        Byte b1=new Byte(str);

        System.out.print("enter another byte no : ");

        str=br.readLine( );

        Byte b2=Byte.valueOf(str);

        int n=b1.compareTo(b2);

        if(n==0)

            System.out.println("both same ");

        else

            if(n>0)
```

```
        System.out.println(b1+"is bigger than"+b2);

    else

        System.out.println(b1+"is lesser than"+b2);

    }

}
```

D:\prr\Core java>javac Bytes.java

D:\prr\Core java>java Bytes

enter a byte no : 34

enter another byte no : 56

34is lesser than56

3) Integer Class: The integer class wraps a value of the primitive type 'int' in a object. An object of type integer contains a single field whose type is int.

Constructors:

- 1) Integer(int num)
- 2) Integer(String str)

Methods:

1) int intValue()

Returns the value of invoking object as an 'int'.

2) int compareTo(Integer obj)

Compare the numerical value of the invoking object of 'obj'. Returns 0, ve or +ve value.

3) static int parseInt(String str) throws NumberFormatException

Returns int equivalent of the string str.

Ex: String str="Hyderabad";

```
Int n=Integer.parseInt(str);
```

4) String toString()

Returns a string from of the invoking object.

5) static Integer valueOf(String str) throws NumberFormatException

Returns an integer object that contain the value shown by str.

6) static String toBinaryString(int i)

Returns a string representation of the integer argument in base 2.

7) static String toHexString(int i)

Returns a string representation of the integer argument in base 6.

8) static String toOctalString(int i)

Returns a string representation of the integer argument in base 8.

//converting into other number system

```
import java.io.*;
class ConvertIntegers
{
    public static void main(String args[])throws IOException
    {
        BufferedReader br=new BufferedReader(new
                                         InputStreamReader(System.in));
        System.out.println("Enter int :");
        String str=br.readLine();
        int n=Integer.parseInt(str);
        System.out.println("in Decimal =" +n);
        str=Integer.toBinaryString(n);
        System.out.println("In Binary =" +str);
        str=Integer.toHexString(n);
        System.out.println("In HexaDecimal =" +str);
    }
}
```

```
        str=Integer.toOctalString(n);

        System.out.println("In Octal =" +str);

    }

}
```

D:\prr\Core java>javac ConvertIntegers.java

D:\prr\Core java>java ConvertIntegers

Enter int :

123

in Decimal =123

In Binary =1111011

In HexaDecimal =7b

In Octal =173

4) Float class: The float class wraps a value of primitive type 'float' in an object. An object of type float contains a single field whose type is float.

Constructors:

1)Float (float num)

2)Float (String str)

Methods:

1) float floatValue()

returns the value of the involving object as a float.

2) double doubleValue()

returns the value of the involving object as a double.

3) int compareTo(Float f)

compares the numeric value of the involving object with that of 'f' returns ., ve.+ve value.

4) static float parseFloat (string str) throwsnumberFormatExecution.

Returns the float equivalent the string str .

5) String toString ()

returns the string equivalent of the involving object.

6) static float valueOf (String str) throwsnumberFormatExecution

.Returns the float object with the value specified by string str.

5) double class: The double class wraps a value of primitive type 'double' in an object. An object of type float contains a single field whose type is double.

Constructors:

1) Double (double num)

2) Double (String str)

Methods:

→1) double floatValue()

returns the value of the involving object as a double.

→2) double floatValue()

returns the value of the involving object as a float.

→3) int compareTo(Double d)

compares the numeric value of the involving object with that of 'd' returns ., ve.+ve value.

→4) static double parseDouble (string str) throws NumberFormatExecution

Returns the double equivalent the string str .

→5) String toString ()

returns the string equivalent of the involving object.

→6) static double valueOf (String str) throws NumberFormatExecution

Returns the double object with the value specified by string str.

6) Math class: The class math contains methods for performing numerical operations.

Methods:

1) static double sin(double arg)

Returns the sine value of arg. Arg is an radians.

2) static double cos(double arg)

Returns the cosine value of arg. Arg is an radians.

3) static double tan(double arg)

Returns the tangent value of arg. Arg is an radians.

4) static double log(double arg)

Returns the natural logarithm value of arg.

5) static double pow(double x,double y)

Returns x to the power of y value.

6) static double sqrt(double arg)

Returns the square root of arg.

7) static double abs(double arg)

Returns the absolute value of arg.

8) static double ceil(double arg)

Returns the smallest integers which is grater or equal to arg.

Ex: Math.ceil(4.5) → is 5.0

Ceil means ceiling. It means up.

9) static double floor(double arg)

Returns the grater integer which is lower or equal to arg.

Ex: Math.floor(4.5) → is 4.0

10) static double min(arg1,arg2)

Returns the minimum value of arg1 & arg2.

11) static double max(arg1,arg2)

Returns the maximum value of arg1 & arg2.

12) static long round(arg)

Returns the rounded value.

Ex: Math.round(4.6) → is 5

13) static double random()

Returns the random numbers between 0&1.

14) static double toRadians(double angle)

Converts angle in degrees in to radians.

15) static double toDegrees(double angle)

Converts angle in radians in to degrees.

// random numbers between 1 & 10

class Rand

{

 public static void main(String args[]) throws InterruptedException

{

 while(true)

{

 double d=10*Math.random();

 int n=(int)d;

 System.out.print(n+"\t");

 if(n==0)

```
        System.exit(0);

        Thread.sleep(3000);

    }

}

}

D:\prr\Core java>javac Rand.java

D:\prr\Core java>java Rand

7      7      8      5      6      2      7      6      1      0
```

Autoboxing and Auto-Unboxing:

Prior to Java SE 5, if you wanted to insert a primitive value into a data structure that could store only Objects, you had to create a new object of the corresponding type-wrapper class, and then insert this object in the collection. Similarly, if you wanted to retrieve an object of a type-wrapper class from a collection and manipulate its primitive value, you had to invoke a method on the object to obtain its corresponding primitive-type value. For example, suppose you want to add an int to an array that stores only references to Integer objects. Prior to Java SE 5, you would be required to “wrap” an int value in an Integer object before adding the integer to the array and to “unwrap” the int value from the Integer object to retrieve the value from the array, as in

```
Integer[] integerArray = new Integer[ 5 ]; // create integerArray
// assign Integer 10 to integerArray[ 0 ]
integerArray[ 0 ] = new Integer( 10 );
// get int value of Integer
int value = integerArray[ 0 ].intValue();
```

Notice that the int primitive value 10 is used to initialize an Integer object. This achieves the desired result but requires extra code and is cumbersome. We then need to invoke method intValue of class Integer to obtain the int value in the Integer object. Java SE 5 simplified converting between primitive-type values and type-wrapper objects, requiring no additional code on the part of the programmer. Java SE 5 introduced two new conversions—the boxing conversion and the unboxing conversion. A **boxing conversion** converts a value of a primitive type to an object of the corresponding typewrapper class. An **unboxing conversion** converts an object of a type-wrapper class to a value of the corresponding

primitive type. These conversions can be performed automatically (called **autoboxing** and **auto-unboxing**). For example, the previous statements can be rewritten as

```
Integer[] integerArray = new Integer[ 5 ]; // create integerArray  
integerArray[ 0 ] = 10; // assign Integer 10 to integerArray[ 0 ]  
int value = integerArray[ 0 ]; // get int value of Integer
```

In this case, autoboxing occurs when assigning an int value (10) to integerArray[0], because integerArray stores references to Integer objects, not int primitive values. Auto-unboxing occurs when assigning integerArray[0] to int variable value, because variable value stores an int value, not a reference to an Integer object. Autoboxing and auto-unboxing also occur in control statements—the condition of a control statement can evaluate to a primitive boolean type or a Boolean reference type.

CHAPTER9

POLYMORPHISM

AND

INHERITENCE

Polymorphism in Java

->It is the concept of defining multiple methods with the same name, but with different functionalities. The methods defined are called 'Overloaded methods' and the concept is called 'method overloading'. Note that, Java language does not support 'Operator Overloading'.

->The difference between the overloaded methods lies in the number or types of parameters that it takes. But, the return types of all overloaded methods must be same.

->The following program illustrates Polymorphism.

```
/* Program for Polymorphism */
```

```
class PolyDemo
{
    void fun1()
    {
        System.out.println("Inside fun1 without parameters ");
    }

    void fun1(int x)
    {
        System.out.println("Inside fun1 with one int parameters ");
    }
}
```

```
System.out.println(x);
}

void fun1(double x)
{
    System.out.println("Inside fun1 with one double parameters ");
    System.out.println(x);
}

void fun1(char x)
{
    System.out.println("Inside fun1 with one char parameters ");
    System.out.println(x);
}

void fun1(int x,int y)
{
    System.out.println("Inside fun1 with two int parameters ");
    System.out.println(x);
    System.out.println(y);
}

void fun1(double x,double y)
{
    System.out.println("Inside fun1 with two double parameters ");
    System.out.println(x);
    System.out.println(y);
}

public static void main(String args[])
{
    PolyDemo p = new PolyDemo();
    p.fun1();
    p.fun1(12.5);
}
```

```

p.fun1(250);
p.fun1('q');
p.fun1(100,200);
p.fun1(12.5,22.5);
} // end of main()
}// end of class

```

Note : -> `println()` is an example for overloaded method since in the `java.io.PrintStream` class multiple `println()` methods are defined each to accept values of corresponding Primitive datatypes.

Rules for Function Overloading:

- overloading can take place in the same class or in the subclass
- overloaded methods **MUST** have a different argument list
- overloaded methods **MAY** change the return type (in case argument list is different)
- overloaded methods **MAY** change the access modifier
- overloaded methods **MAY** throw new or broader checked exceptions
- reference type determines which overloaded method will be used at compile time
- constructors **MAY** be overloaded
- methods adjustment in connection with overloaded method's arguments:
 - you **cannot** widen and then box (`int -> Long`)
 - you **can** box and then widen (`int -> Object, via Integer`)
 - you **can** combine var args with either widening (`byte -> int`) or boxing (`int -> Integer`):
 - widening is over boxing
 - widening is over var args
 - boxing is over var args

-> Polymorphism is implemented in Java Language in two ways. They are:

1. Static Polymorphism.
2. Dynamic Polymorphism.

Static Polymorphism: Here, if we have multiple functions with same name in a class, we can know which function will gets executed when a function is called. 'Function Overloading' is an example for 'Static' or 'Compile-time' polymorphism.

-> But in some cases until we run the program which method statements are executed is not known. Such type of polymorphism is called 'Dynamic' or 'Run-time' Polymorphism. Dynamic Polymorphism is implemented through 'Method Overriding'.

NOTE : `println()` is an example for overloaded methods which can accept any type of data and prints on the console.

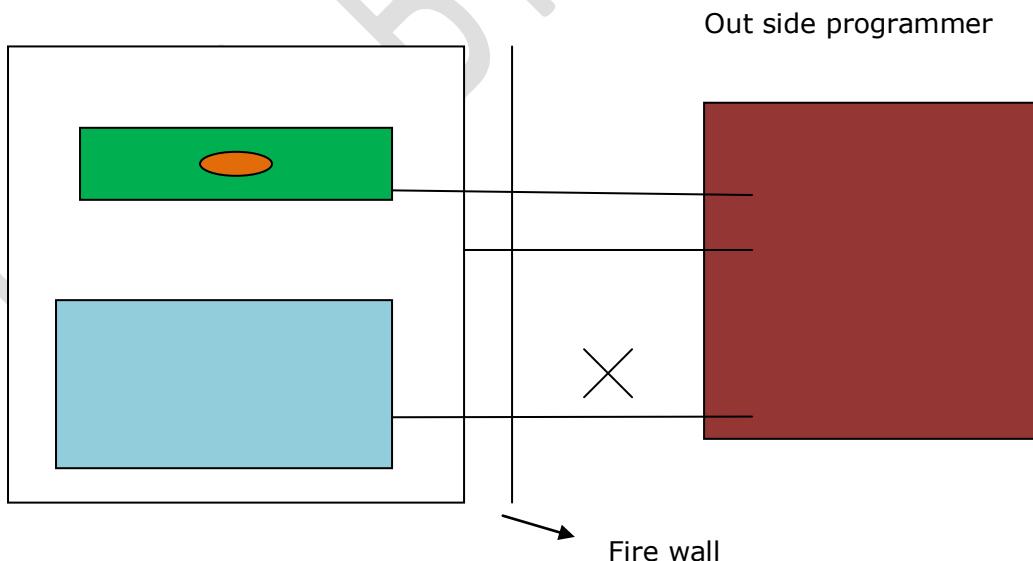
What happens when we call an overloaded function ?

- >First, the compiler checks whether there is any function defined to accept actual data types. If found, it gets executed.
- > If not found, then it checks whether there is any function whose parameters belong to same datatype family. If found, it will be executed.
- > If not found, it will go for "alternate function" and executes it. But, If there are multiple alternate functions in a class, it becomes ambiguous for the JVM to load and execute which overloaded function. So, the program is terminated.

Inner classes:

Inner class is a class written in another class. It is a security mechanism.

We can use private in inner class. Private keyword can be used before inner class only. Fire wall means optical that checks the authorization of user.



```
// inner class demo
```

```
class BankAcct
```

```
{
```

```
CALL:9010990285/7893636382
```

```
private double bal;

BankAcct(double b)

{

    bal=b;

}

void start(double r)

{

    Interest in=new Interest(r);

    in.calculateInterest( );

}

private class Interest

{

    private double rate;

    Interest(double r)

    {

        rate=r;

    }

    void calculateInterest( )

    {

        System.out.println("Balance = "+bal);

        double interest=bal*rate/100;

        System.out.println("interest = "+interest);

        bal+=interest;

        System.out.println("New Balance = "+bal);

    }

}
```

```
        }

    }

class InnerDemo

{

    public static void main(String args[])

    {

        BankAcct account=new BankAcct(20000);

        account.start(7.5);

    }

}
```

D:\prr\Core java>javac InnerDemo.java

D:\prr\Core java>java InnerDemo

Balance = 20000.0

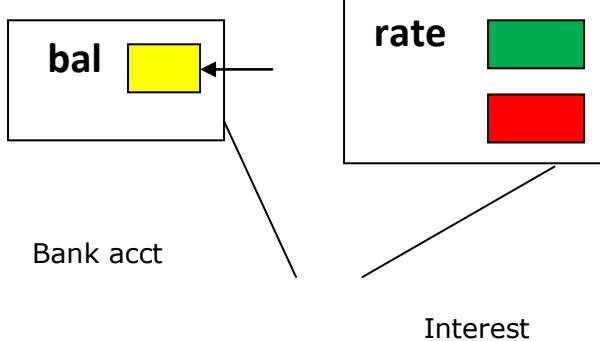
interest = 1500.0

New Balance = 21500.0

Important points about Inner classes:

- 1) Inner class is a written within another class.
- 2) It is a security mechanism.
- 3) Inner class is hidden in outer class from other classes.
- 4) Only inner class can be 'private'
- 5) An object to inner class can not be created in any other class.
- 6) An object to inner class can be created only in its outer class.
- 7) Outer class object and inner class objects are created separately in memory.
- 8) Outer class members are available to inner class.

- 9) Inner class object contains an additional invisible field 'this \$0' that contains outer class reference.



- 10) if same names are used for members ,then outer class members can be referenced in inner class as: outer class.this.member;

Ex: bankAcct.this.bal.

Inner class members are referenced as: this.member

Ex: this.bal

Inheritance in Java:

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

Inheritance is the property that a class gets the properties of another class. The class which gets the properties is called 'Super class' or 'Base class' and the class which provides the properties is called 'Sub class' or 'Derived Class'. Reusability and Extensibility are the major advantages of Inheritance.

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

Types of Inheritance:

Java Supports 3 types of Inheritance. They are:

- > Single Inheritance [single base, single derived]
- > Multi-level Inheritance [A derived class derived by another derived class].
- > Hierarchical Inheritance [Single base, multiple derived].

Note: Java does not support the concepts of Multiple and Hybrid Inheritance.

-> Java supports a keyword called 'extends' to implement inheritance. Its syntax is:

```
class Base-class-name  
{  
---  
}  
  
class Derived-class-name extends Base-class-name  
{
```

}

The following program illustrates Single Inheritance:

```
/* Program for Single Inheritance */

class Square

{
    int l;

    Square()
    {
        System.out.println("Inside Constructor of Square");

        l = 100;
    }

    void area1()
    {
        int area = l * l;

        System.out.println("Area of Square = " + area);
    }
}

} // End of Square class

class Rectangle extends Square

{
    int b;

    Rectangle()
    {
        System.out.println("Inside Constructor of Rectangle");

        b = 200;
    }

    void area2()
    {
        int area = l * b;
    }
}
```

```

        System.out.println("Area of Rectangle = " + area) ;
    }

} // End of Rectangle class

class SingleInheritance
{
    public static void main(String args[])
    {
        Rectangle r = new Rectangle();
        r.area1();
        r.area2();
    } // end main()
} //end class

```

What happens when a sub-class object is created?

-> While creating the object of sub-class, the JVM checks whether the class is extending any class or it. If extending, it first creates the object of super class and then the sub-class object is created. That means: **"Sub-class objects are created only after super class objects area created".**

NOTE: The super-most class of any Java class is the "**java.lang.Object**" class.

IS-A Relationship:

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

```

public class Animal{
}

public class Mammal extends Animal{
}

public class Reptile extends Animal{
}

public class Dog extends Mammal{
}

```

Now based on the above example, In Object Oriented terms following are true:

- Animal is the super class of Mammal class.
- Animal is the super class of Reptile class.
- Mammal and Reptile are sub classes of Animal class.

- Dog is the subclass of both Mammal and Animal classes.

Now if we consider the IS-A relationship we can say:

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

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

The instanceof Keyword:

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

```
interface Animal{}
class Mammal implements Animal{}

class Dog extends Mammal{
    public static void main(String args[]){
        Mammal m = new Mammal();
        Dog d = new Dog();

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

HAS-A relationship:

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

Lets us look into an example:

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

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

In Object Oriented feature the users do not need to bother about which object is doing the real work. To achieve this, the Van class hides the implementation details from the users of the Van class. SO basically what happens is the users would ask the Van class to do a certain

action and the Vann class will either do the work by itself or ask another class to perform the action.

super keyword in Java:

-> Java language a keyword called 'super' which can be used for 2 purposes.They are:

- 1) To call the constructor of super class from sub-class constructor.
- 2) To call the members of super class from sub-class members.

->The following program illustrates the super keyword and multi-level Inheritance.

```
/* Program for Multi-level Inheritance and to demonstrate super keyword */
```

```
class Square

{
    int l;

    Square(int x)

    {
        l=x;
    }

    void area()

    {
        System.out.println("Area of Square = " + l*l);
    }
} // End of class Square

class Rectangle extends Square

{
    int b;

    Rectangle(int x,int y)

    {
        super(x); /*passes x to constructor of Square class */
        b=y;
    }

    void area()
```

```
{  
super.area(); // calls area() of Square class  
System.out.println("Area of Rectangle = " + l*b);  
}  
} //End of class Rectangle  
  
class Cube extends Rectangle  
{  
int h;  
Cube(int x,int y, int z)  
{  
super(x,y);  
h=z;  
}  
void area()  
{  
super.area(); // calls area() of Rectangle class  
System.out.println("Volume of cube= " + l*b*h);  
}  
} //End of class Cube  
  
class SuperDemo2  
{  
public static void main(String args[])
{
    Cube c1 = new Cube(10,20,30);
    c1.area();
}
}
```

Note:

- 1) The statement 'super()' should be the first statement in the constructor of a sub class.

2) By default, every class constructor will have a statement called 'super()' as the first statement in its constructor. Because of this statement only, the super class objects are created before sub class objects are being created.

The following program demonstrates Hierarchical Inheritance

```
//Program fro Hierarchical Inheritance

class Square
{
    double l=10.0;
}

class Rectangle extends Square
{
    double b=20.0;
    void area()
    {
        System.out.println("Area of Rectangle= " + l*b);
    }
}

class Circle extends Square
{
    void area()
    {
        System.out.println("Area of circle= " + 3.14*l*l);
    }
}

class IDemo6
{
    public static void main(String args[])
    {
        Rectangle R=new Rectangle();
```

```
R.area();  
Circle C=new Circle();  
C.area();  
}  
}
```

NOTE: In Java, **for the reference of a super class, we can assign the object of a sub-class.** With that reference, we can call the methods defined in super class only. When a method is called with that reference, the JVM first checks whether that method is defined in super class or not. If defined, it moves to sub most object and execute that class method. If not available in the sub-class, it moves to next immediate super class and repeats the procedure until the method is found.

->The following program demonstrates super-class reference and sub-class concept.

```
/** Super class reference, subc-class object */  
  
class A  
{  
    void fun1()  
    {  
        System.out.println("Inside fun1 of A");  
    }  
    void fun2()  
    {  
        System.out.println("Inside fun2 of A");  
    }  
    void fun3()  
    {  
        System.out.println("Inside fun3 of A");  
    }  
}//end of class A  
  
class B extends A  
{  
    void fun2()
```

```
{  
System.out.println("Inside fun2 of B");  
}  
  
void fun4()  
{  
System.out.println("Inside fun4 of B");  
}  
}  
}//end of class B  
  
class C extends B  
{  
void fun3()  
{  
System.out.println("Inside fun3 of C");  
}  
void fun5()  
{  
System.out.println("Inside fun5 of C");  
}  
}  
}  
  
class Inheritance3  
{  
public static void main(String args[])  
{  
A a1=new C();  
a1.fun1(); // prints fun1() of A  
a1.fun2(); // prints fun2() of B  
a1.fun3(); // prints fun3() of C  
/* a1.fun4(); // cannot be called since it is not defined in class A  
a1.fun5(); // cannot be called since it is not defined in class A */
```

```
}
```

```
}// end of class Inheritance3
```

Polymorphism: It represents many forms. If a method performs several tasks, it is called polymorphism. They are 2 types

- 1) Dynamic polymorphism 2) Static polymorphism

1) Dynamic polymorphism:

The polymorphism exhibited at run time is called dynamic polymorphism. In this dynamic polymorphism a method call is linked with method body at the time of running the program by JVM. This is also known as dynamic binding or run time polymorphism.

EX:

Method over loading:

Writing 2 or more methods with the same name, but with a difference in the method signatures is called method over loading.

In method over loading JVM understand in the method is called depending upon the difference in the method signature. The difference may be due to the following.

- 1) There is a difference in the no. of parameters.

```
void add(int a,int b)
```

```
void add(int a,int b,int c)
```

- 2) There is a difference in the data types of parameters.

```
void add(int a,float b)
```

```
void add(double a,double b)
```

- 3) There is a difference in the sequence of parameters.

```
void swap(int a,char b)
```

```
void swap(char a,int b)
```

Any one of the above differences will make method signature to be different, hence JVM can identifies methods uniquely.

→ Method over loading is an example for dynamic polymorphism.

```
// sum of 2 numbers & sum of 3 numbers

class Sample

{

    void add(int a,int b)

    {

        System.out.println("sum of two="+(a+b));

    }

    void add(int a,int b,int c)

    {

        System.out.println("sum of three="+(a+b+c));

    }

}

class Poly

{

    public static void main(String[] args)

    {

        Sample s=new Sample( );

        s.add(20,25);

        s.add(20,25,30);

    }

}
```

D:\prr\Core java>javac Poly.java

CALL:9010990285/7893636382

```
D:\prr\Core java>java Poly
```

```
sum of two=45
```

```
sum of three=75
```

Q) What is method signature?

A) Method name & its parameters is called method signature

Method overriding: writing 2 or more methods in super & sub classes with same name & same signatures is called method overriding. In method overriding JVM executes a method depending on the data type of the object.

→ Method overriding is an example for dynamic polymorphism

Rules for overriding the methods:

1. applies ONLY to inherited methods
2. is related to polymorphism
3. object type (NOT reference variable type) determines which overriden method will be used at runtime
4. overriding method MUST have the same argument list (if not, it might be a case of overloading)
5. overriding method MUST have the same return type; the exception is covariant return (used as of Java 5) which returns a type that is a subclass of what is returned by the overriden method
6. overriding method MUST NOT throw new or broader checked exceptions, but MAY throw fewer or narrower checked exceptions or any unchecked exceptions
7. abstract methods MUST be overridden
8. final methods CANNOT be overridden
9. static methods CANNOT be overridden
10. constructors CANNOT be overridden

//dynamic polymorphism

```
class One
```

```
{
```

```
    void calculate(double x)
```

```
{
```

```
System.out.println("Square value =" +(x*x));  
}  
}  
  
class Two extends One  
{  
  
    void calculate(double x)  
    {  
        System.out.println("Square root =" +Math.sqrt(x));  
    }  
}  
  
class Poly1  
{  
  
    public static void main(String args[])  
    {  
        Two t=new Two();  
        t.calculate(16);  
    }  
}  
  
D:\prr\Core java>javac Poly1.java  
  
D:\prr\Core java>java Poly1  
  
Square root =4.0
```

/* Program for Dynamic Polymorphism */

```
class A  
{
```

```
void fun1()
{
    System.out.println("Inside fun1 of A");
}

void fun2()
{
    System.out.println("Inside fun2 of A");
}

}

class B extends A
{
    void fun1()
    {
        System.out.println("Inside fun1 of B");
    }

    void fun2()
    {
        System.out.println("Inside fun2 of B");
    }
}

class C extends B
{
    void fun1()
    {
        System.out.println("Inside fun1 of C");
    }

    void fun2()
    {
        System.out.println("Inside fun2 of C");
    }
}
```

```
}

}

class DynamicPolymorphism

{

public static void main(String args[])

{

A a1;

int n=Integer.parseInt(args[0]);

if(n<=5)

a1=new A();

else if(n>5 && n<10)

a1=new B();

else

a1=new C();

a1.fun1();

a1.fun2();

} // end of main()

} // end of class
```

Q) What is the difference between method over loading & method overriding?

A) **Method over loading:** Writing 2 or more methods with the same name, but with a difference in the method signatures is called method over loading. In method over loading JVM understand in the method is called depending upon the difference in the method signature.

Method overriding: writing 2 or more methods in super & sub classes with same name & same signatures is called method overriding. In method overriding JVM executes a method depending on the data type of the object.

Achieving method overloading & method overriding using instance methods is an example of dynamic polymorphism.

Achieving method overloading & method overriding using instance methods is an example of dynamic polymorphism.

2) Static polymorphism: The polymorphism exhibited at compile time is called Static polymorphism. Here the compiler knows which method is called at the compilation. This is also called compile time polymorphism or static binding.

Achieving method overloading & method overriding using

1) private 2) static 3) final methods, is an example of Static polymorphism.

Note: A final method is a method written in a final class. A class is declared as a final is called final class.

Ex: final class A

Note: final key word before class name prevents inheritance. We can create sub classes to a final class.

Ex: final class A

class B extends A // invalid

→ We can not override final methods, only final methods overloading.

Converting 1 data type into another data type is called casting.

"final" keyword in Java:

-> Java Language supports a special keyword called 'final' which can be used for 4 purposes .They are:

1) Declaring variable as final:

->If a variable is declared as 'final', then, that variable becomes a constant. The syntax is:

final type var-name = value;

eg:

final float PI = 3.1428F;

2) Declaring a method as final:

-> If we use 'final' with a method , that method cannot be overridden. The syntax is :

final return-type method-name([params]) { }

3) class as final :

-> If we define a class as 'final' , that class cannot be extended. The syntax is :

final class class-name

{

```
---  
}
```

- 4) The fourth use of 'final' keyword is that we can use it as a special block in exception-handling as finally block. The syntax is :

```
finally  
{  
---  
}
```

Type casting:

Q) What is difference between primitive & advanced data types?

- A) 1) Casting can be used to convert one primitive data type in to another primitive data type.
- 2) Casting can be used to convert one advanced data type into another advanced data type.
- 3) Casting can not be used to convert a primitive data type into an advanced data type and vice versa (means reverse also).

For this purpose we can use wrapper classes.

Casting primitive data types:

- a) **Widening:** Casting a lower data type into a higher data type is called widening.

EX: char, byte, short, int, long, float, double

Lower ← →higher

```
i) char ch='A';  
    int n=(int)ch;  
  
ii) int num=15;  
    float f=(float)num;
```

In widening no digits or precision are lost.

- b) **Narrowing:** Converting a higher data type into lower type is called narrowing.

EX: i) int n=66;

```
Char ch=(char)n;  
ii) double d=12.1234;  
int n=(int)d;
```

In narrowing precision or digits are lost. This is called explicit casting.

Casting advanced data types:

We can cast advanced data types provide there is relationship between the classes by the way of inheritance.

//Casting advanced data types:

```
class One  
{  
    void show1( )  
    {  
        System.out.println("Super class");  
    }  
}  
  
class Two extends One  
{  
    void show2( )  
    {  
        System.out.println("Sub class");  
    }  
}  
  
class Cast  
{  
    public static void main(String args[])
```

```
{  
    // super class reference to refer to super class object  
  
    One o;  
  
    o=new One( );  
  
    o.show1( );  
  
}  
}
```

D:\prr\Core java>javac Cast.java

D:\prr\Core java>java Cast

Super class

In super reference is used to refer to super class object. The programmer can refer to only super class members but not the sub class members

class Cast

```
{  
    public static void main(String args[]){  
        // sub class reference to refer to sub class object  
  
        One o=new Two( );  
  
        Two t=(Two)o;  
  
        t.show1( );  
  
        t.show2( );  
    }  
}
```

D:\prr\Core java>javac Cast.java

D:\prr\Core java>java Cast

Super class

Sub class

In sub class reference is used subclass object; the programmer can access all the members of both the super class& sub class.

class Cast

```
{  
    public static void main(String args[]){  
        // super class reference to refer to sub class object  
        One o;  
        o=new Two();  
        o.show1();  
    }  
}
```

D:\prr\Core java>javac Cast.java

D:\prr\Core java>java Cast

Super class

Q) What is generalization & specialization?

A) Moving back from sub class to super class is called generalization. Casting a sub class type into a super class is called up casting or widening.

Coming down from super class to sub class is called specialization. Casting a super class type into sub class type is called narrowing or down casting.

In widening we can access super class methods. In widening we can access sub class methods. In widening, we can not access sub class methods unless the override super class methods.

class Cast

CALL:9010990285/7893636382

```
{  
    public static void main(String args[])  
    {  
        // sub class reference to refer to super class object  
        One o=new Two();  
  
        Two t=(Two)o;  
  
        t.show1();  
  
        t.show2();  
    }  
}
```

D:\prr\Core java>javac Cast.java

D:\prr\Core java>java Cast

Super class

Sub class

Narrowing using super class object can not access either super class methods or sub class methods.

Narrowing using sub class object will make the programmer to access are the members of super class as well as sub class.

Q) What is widening & narrowing?

A) Widening: Casting a lower data type into a higher data type is called widening. In widening no digits or precision are lost. Casting a sub class type into a super class is called an up casting or widening. In widening we can access super class methods. In widening we can access sub class methods. In widening us can not access sub class methods unless the override super class methods.

Narrowing: Converting a higher data type into lower type is called narrowing. In narrowing precision or digits are lost. This is called explicit casting. Casting a super class type into sub class type is called narrowing or down casting. Narrowing using super class object can not

access either super class methods or sub class methods. Narrowing using sub class object will make the programmer to access are the members of super class as well as sub class.

Cloning:

Cloning is a technology to obtain exact copy of a plant, a bird an animal and a human being.

Cloning in java:

Obtaining bitwise exact copy of an object is called cloning.

- 1) Shallow cloning: In this, any modifications to the cloned object, will also modify the original object.
- 2) Deep cloning: in this cloning any modifications to the cloned object will not affect the original object.

EX:

// cloning example

```
class Employee implements Cloneable

{
    int id;
    String name;
    Employee(int i,String s)
    {
        id=i;
        name=s;
    }
    void display( )
    {
        System.out.println("Id = "+id);
        System.out.println("Name = "+name);
    }
}
Employee myClone( )throws CloneNotSupportedException
```

```
// (or) protected Object Clone( ) throws CloneNotSupportedException  
{  
    return(Employee)super.clone( );  
    // (or) return super.clone( );  
}  
}  
  
class CloneDemo  
{  
    public static void main(String args[]) throws CloneNotSupportedException  
    {  
        Employee e1=new Employee(10,"sudheer");  
        e1.display( );  
        Employee e2=e1.myClone( );  
        // (or) Employee e2=(Employee)e1.Clone( );  
        e2.display( );  
    }  
}
```

D:\prr\Core java>javac CloneDemo.java

D:\prr\Core java>java CloneDemo

Id = 10

Name = sudheer

Id = 10

Name = sudheer

→ Cloneable method does not contain any interface. A tagging interface or a marking interface is an interface without any methods or with zero abstract methods

Interface indicates a special purpose for the object of the class.

Ex: Cloneable, Serialization.

→ Getting back object from the file is called Serialization. Storing a object in to a file is called a Serialization.

JAVA BY SATEESH

CHAPTER10

INTERFACES

Interfaces in Java:

->Interfaces are the structures in Java using which “Dynamic Binding” is implemented. An interface is similar to a class except that every method inside a class is provided bodies while no method of an interface must be defined with bodies. That’s why classes are called ‘Fully implemented Structures’ while the interfaces are called ‘Fully unimplemented Structures’.

->The syntax for creating an interface is as follows:

```
interface interface-name
```

```
{
```

```
variable-declarations;
```

```
method declarations;
```

```
}
```

-> Every data member of the interface should be declared as 'final'. Every member of an interface should be declared 'public'. To provide bodies for the methods of an interface, we have to create a class that implements the interface as follows:

```
class class-name implements interface-1,interface-2,...
```

```
{  
    // define bodies for methods of interface  
}
```

->The following program illustrates the use of an interface.

```
// Program for interfaces  
  
interface Circle  
  
{  
    public final float PI = 3.1428F;  
  
    public void area();  
}  
  
class InDemo1 implements Circle  
  
{  
    public void area()  
    {  
        float r = 10.0F;  
  
        float area = PI * r * r;  
  
        System.out.println("Area of Circle = " + area);  
    }  
}
```

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

```
{
```

```
InDemo1 in = new InDemo1();
```

```
CALL:9010990285/7893636382
```

```
in.area();

} // end of main

} // end of class
```

Note:

- 1) "implements" keyword does not mean "Inheritance" because "implements" means providing properties while "extends" means getting the properties.
- 2) For an interface, we can create reference. But, we cannot create object. Because, to create reference, .class file is required, but to create object class is required
- 3) To the reference of an interface, we can assign object of a class which is implementing that interface. Through this reference, we can call the methods of the interface only but not of the implementing class. The following program illustrates this concept:

/ Program to assign the object of a class to the reference of an interface */**

```
interface Xyz

{

    public void funX();

    public void funY();

}

class X implements Xyz

{

    public void funX()

    {

        System.out.println("Inside funx");

    }

    public void funY()

    {

        System.out.println("Inside funY");

    }

}
```

```
}

void fun1()

{

System.out.println("Inside fun1");

}

}// end of class

class InDemo2

{

public static void main(String args[])

{

Xyz x1=new X(); //X implements Xyz

x1.funX();

x1.funY();

//x1.fun1(); ->Error

}// end main

}// end class
```

->Generally, the programmers think that through the concept of interfaces Java supports Multiple Inheritance. The following program is an example for that implements this concept.

/ Program that demonstrates Multiple Inheritance in Java */**

```
interface Xyz

{

public void funX();

public void funY();

}

interface Abc
```

```
{  
    public void funA();  
  
    public void funB();  
}  
  
interface Mno extends Xyz,Abc  
{  
  
    public void funM();  
  
    public void funN();  
}  
  
class Sample implements Mno  
{  
  
    public void funX()  
    {  
        System.out.println("Inside funX");  
    }  
  
    public void funY()  
    {  
        System.out.println("Inside funY");  
    }  
  
    public void funA()  
    {  
        System.out.println("Inside funA");  
    }  
  
    public void funB()  
    {  
        System.out.println("Inside funB");  
    }  
}
```

```
System.out.println("Inside funB");

}

public void funM()

{

    System.out.println("Inside funM");

}

public void funN()

{

    System.out.println("Inside funN");

}

}// end of class

class InDemo3

{

    public static void main(String args[])

    {

        Sample s = new Sample();

        s.funX();

        s.funY();

        s.funA();

        s.funB();

        s.funM();

        s.funN();

    }// end of main

}// end of class
```

->in the above example, because the interface Mno is extending two interfaces (Abc, Xyz), programmers call it Multiple Inheritance. But, the methods of Mno, Abc and Xyz are provided bodies by the class Sample. That is, Mno is not getting anything from Abc and Xyz instead bodies for the methods of all the interfaces is provided by Sample class.

->Now let us see a program where we can create object for an interface.

/* Program for Creating object for an Interface */

```
interface Xyz
{
    public void funX();
    public void funY();
}

class InDemo
{
    public static void main(String args[])
    {
        Xyz x1 = new Xyz() {
            public void funX()
            {
                System.out.println("Inside funX");
            }
            public void funY()
            {
                System.out.println("Inside funY");
            }
        };
        x1.funX();
    }
}
```

```
x1.funY();  
}  
} //end of class
```

-> In the above program, it seems that we are creating object for the interface Xyz using 'new' operator. But, it is some class that is providing bodies for methods of the interface Xyz and its name is unknown. Such classes which implements an interface and without having a name are called "Anonymous Inner classes". Most of the classes in Java API are defined as Anonymous Inner classes.

Q) What kind of methods interfaces having?

Interface contains only public abstract methods

Ex:

```
interface First{  
    public abstract int add();  
}
```

By default a class method is package access method, but by default an interface method is public abstract.

Important points to remember:

- 1)** We can access interface data by using interface name or by using interface reference pointing null.
- 2)** We cannot change interface data because it is final by default
- 3)** An interface can extend another interface
- 4)** An interface can extend multiple interfaces
- 5)** An interface cannot extend a class
- 6)** An interface cannot implement a class
- 7)** An interface cannot implement another interface
- 8)** A class can extend only one class
- 9)** A class cannot extend multiple classes
- 10)** A class can implement one or more interfaces

Abstract Classes in Java:

- > The third type of structures supported by Java Language is an abstract class which is called a partially-implemented and partially-unimplemented structure.
- > The main difference between a class and an abstract class is that in a class, all methods will have implementations while in an abstract class, some methods are implemented and some are unimplemented. The unimplemented methods of abstract classes are called 'abstract methods' and must be declared 'abstract'.
- > This class is defined with a keyword called 'abstract' and may contain both variables and methods like a class. The unimplemented methods of an abstract class will be provided bodies by a class which extends it.

Note: Like interfaces, we cannot create object of a class directly. The class which extends the abstract class, through its object only we can call abstract class members

-> If a class that is extending an abstract class doesn't want to provide methods for the abstract class, it should also be declared as 'abstract'.

-> The following program demonstrates abstract classes:

```
// Program to implement abstract classes
```

CALL:9010990285/7893636382

```
abstract class abs1

{
    void fun1()
    {
        System.out.println("Inside fun1");
    }

    void fun2()
    {
        System.out.println("Inside fun2");
    }

    abstract void fun3(); //abstract method
    abstract void fun4(); //abstract method
}

class A extends abs1

{
    void fun3()
    {
        System.out.println("Inside fun3");
    }

    void fun4()
    {
        System.out.println("Inside fun4");
    }

    void fun5()
    {
```

```
System.out.println("Inside fun5");

}

}// end of class A

class AbsDemo1

{

public static void main(String args[])

{

A a1 = new A();

a1.fun1();

a1.fun2();

a1.fun3();

a1.fun4();

a1.fun5();

} // end main

}// end class
```

NOTE:

- 1) Like interfaces, for the reference of an abstract class, we can assign the object of the class that extends that class. With this reference, we can call the methods of the abstract class only.
- 2) If a class that is extending an abstract class does not want to provide bodies for the methods of abstract class, then, it should also declared as 'abstract'.
- 3) We can define a class as abstract class without abstract methods.
- 4) We can create abstract class reference but we can't create abstract class object.

Important points to remember about abstract class:

- 1) Abstract class can contain static data members and non static data members.

- 2) Abstract class can contain static methods and non static methods
- 3) Abstract class can contain abstract methods and normal methods
- 4) Abstract class can contain constructors
- 5) Abstract class constructors are called by sub class constructors.
- 6) Abstract class can extend another normal classes or abstract classes
- 7) Abstract class can implement other interfaces
- 8) An abstract class cannot be declared as final class
- 9) An abstract method cannot be declared as final method
- 10) An abstract method cannot be declared as private method

Q) Difference Interface & Abstract Class?

1. Abstract classes may have some executable methods and methods left unimplemented. Interface contains no implementation code.
2. An abstract class can have nonabstract methods. All methods of an Interface are abstract.
3. An abstract class can have instance variables. An Interface cannot.
4. An abstract class can define constructor. An Interface cannot.
5. An abstract class can have any visibility: public, private, protected. An Interface visibility must be public (or) none.

Q) In how many ways can you create on object in java?

A) Ways of creating on object:

- 1) Using new operator

```
Employee obj=new Employee();
```

- 2) using factory method

```
NumberFormat obj=new NumberFormat.getInstance();
```

- 3) using newInstance() method

```
Class c=Class.forName("Employee");
```

For name method will store employee that class name in an object

This is represented by c.

- 4) Using cloning()

Cloning means creating bitwise exact copy of an existing object.

JAVA BY SATEESH

CHAPTER12

EXCEPTION HANDLING

Exception Handling:

-> Exception Handling is the procedure of making the Exception class object not reaching the JVM. Java language supports a number of classes each to represent a separate logical error. The super most classes of all exceptions is the "Throwable class" to which there are two sub-classes : Exception and Error classes. All the exception classes are the sub-classes of Exception class.

->To handle exceptions, Java defined two keywords: try and catch blocks. The statements that are proven to generate exceptions must be defined in the try block. When an exception is raised, that exception class object must be handled in the corresponding catch block. The syntax of try-catch blocks is shown as below:

```
try
{
    // statements
}
catch(Exception-type e)
{
    //statements
}
```

->The following program demonstrates Exception Handling.

```
/* Program to handle ArithmeticException */

class Exception1

{

public static void main(String args[])

{

int a,b,c;

a=Integer.parseInt(args[0]);

b=Integer.parseInt(args[1]);

try

{

c = a / b;

System.out.println("Result is : " + c);

}

catch(ArithmeticException e)

{

System.out.println("Arithmetic Exception Caught");

System.out.println("Division by zero is not possible");

}//catch

}//main

}//class
```

-> In the above program, if the ArithmeticException is raised, it is handled by the corresponding catch block.

NOTE: A single try block may generate multiple exceptions. Hence, to handle each Exception-type, we have to define separate catch blocks. It is important to note that a try block can have multiple catch blocks. But, a catch block must associate only one try block.

->The following program illustrates a try block with multiple catch blocks.

/* Program to define multiple catch blocks under a try block */

```
class Exception2

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

        try{
            a[5]=a[2] / (a[1]-20);

            System.out.println("result= " + a[5]);
        }

        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println(e);
        }

        catch(ArithmaticException e)
        {
            System.out.println(e);
        }
    }//main

}//class
```

->In the above program, if a try block is defined to generate multiple exceptions, which exception object is created first, its corresponding catch block will be executed. Then, the program will be terminated.

->Suppose that if a try block is defined to generate more number of exceptions, defining corresponding catch blocks will increase the program size. To avoid this problem, to handle

any exception type, we can define a single catch block where we define the reference of the super class of Exception classes "Exception class".

->The following program demonstrates this concept:

```
/* Program to define a catch block that handles multiple Exceptions */

class Exception3

{

void fun1()

{

int a[]={10,20,30,40,50};

try{

a[5]=a[2] / (a[1]-20);

System.out.println("result= " + a[5]);

}

catch(Exception e)

{

System.out.println(e);

}

}// end of fun1()

public static void main(String args[])

{

Exception3 e1 = new Exception3();

E1.fun1();

}//main

}//class
```

->In the above program,the try block in fun1() is defined to generate ArithmeticException and ArrayIndexOutOfBoundsException. But, the first exception that will raise is ArithmeticException so the corresponding catch block will gets executed.

"throws" keyword :

-> Generally if a function is defined to generate an exception, we have to handle the exception in that function only. Suppose assume that a function does not wants to handle exceptions inside it. Then, they can pass the exception class objects to the functions which call them. This can be done by using a special keyword called "throws". If a function is defined to throw an exception, then the calling function must handle that exception inside it.

-> The following program demonstrates the use of "throws" keyword.

/* Program to demonstrates throws keyword */

```
class Sample

{
    void fun(int x,int y) throws ArithmeticException
    {
        int a,b,c;
        a = x;
        b = y;
        c = a / b;
        System.out.println(" c= " + c);
    }
}
```

```
class ThrowDemo
```

```
{
    public static void main(String args[])
    {
        int a = Integer.parseInt(args[0]);
    }
}
```

```
int b = Integer.parseInt(args[1]);  
  
Sample s = new Sample();  
  
try {  
  
    s.fun(10,0);  
  
}  
  
catch(ArithmeticException e)  
  
{  
  
    System.out.println(e);  
  
}  
  
}// end main  
  
} // end class
```

->In the above program, fun1() is defined to throw ArithmeticException. Hence, main() method is handling it within its try-catch block. If main() method also doesn't want to handle that exception, we can define "throws" with main(). Then that object is passed to JVM and the program is terminated abnormally.

Simple v/s Serious Logical errors :

->Logical errors can be classified into Simple and serious logical errors. If a logical error is a negligible one, we call it a "Simple Logical Error". The Exception classes that derive RunTimeException class represent Simple Logical Error. If the logical errors are non-negligible, they are "Serious Logical Errors". The classes defined under java.lang.Error class and Exception class(Except RunTimeException) represent serious logical errors.

Checked and Unchecked Exceptions :

->Any Exception class that does not extends java.lang.RuntimeException is referred as "Checked Exceptions". Checked Exceptions cannot be handled in a program so we have to use "throws" keyword with them.

->Any Exception class that extends java.lang.RuntimeException is referred as "Unchecked Exceptions". Unchecked Exceptions must be handled in try-catch blocks and must not be defined with "throws" keyword".

Creating user-defined Exceptions :

->User-defined exceptions are created in the form of classes that extend java.lang.Exception class.The following program demonstrates user-defined exceptions.

```
/* Consider a Banking System. The customers while maintaining the accounts must open some Minimum Balance(say Rs.500). In this Example, we create an Exception called InvBalExcepti on whose object is thrown when a user violates Minimum Balance rule */

class InvBalException extends Exception

{

    public String toString()

    {

        String s = " Invalid Amount. Balance should be minimum Rs.500";

        return s;

    }

}

class Account

{

    public float withdraw(float bal,float amount) throws InvBalException

    {

        if( bal - amount < 500)

            throw new InvBalException();

        return bal-amount;

    }

}

class MyException

{

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

```
{  
Account a1 = new Account();  
  
try {  
  
float bal = a1.withDraw(5000,4501);  
  
System.out.println("Your new balance = " + bal);  
  
}  
  
catch(InvBalException e)  
{ System.out.println(e); }  
  
}//main  
  
}//class
```

finally Block:

->Generally if an Exception is raised in a program, its object is handled in the corresponding catch block. Once a catch block is executed, the program will be terminated.

->But,in some cases, irrespective whether an exception is raised or not raised, we want to execute some set of statements mandatory. Such statements can be defined under a finally block.The following program demonstrates the use of “finally” block.

```
//Program that demonstrates “finally” block
```

```
class FinallyDemo  
{  
public static void main(String args[])  
{  
int a[] = {10,20,30,40,50};  
  
try {  
a[4] = a[2] / (a[1] - 10);  
  
System.out.println(a[4]);  
}
```

```
}

catch(Exception e)

{

System.out.println(e);

}

finally

{

System.out.println("This block will definitely execute");

}

}//main

}//class
```

Q) What is the difference between final, finally, finalize?

Final: - When we declare a sub class a final the compiler will give error as "cannot subclass final class" Final to prevent inheritance and method overriding. Once to declare a variable as final it cannot occupy memory per instance basis.

- Final class cannot have static methods
- Final class cannot have abstract methods
- Final class can have only a final method.

Finally: - Finally create a block of code that will be executed after try catch block has completed. Finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement match the exception. Any time a method is about to return to the caller from inside try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also execute.

Using **System.exit ()**; in try block will not allow finally code to execute

Finalize: - sometimes an object need to perform some actions when it is going to destroy, if an object holding some non-java resource such as file handle (or) window character font, these resources are freed before the object is going to destroy any cleanup processing before the object is garbage collected.

CHAPTER13

JAVA.LANG PACKAGE

Java.lang Package Interview Questions

1 what is the base class of all classes?

Ans: java.lang.Object

2 what do you think is the logic behind having a single base class for all classes?

Ans: 1. casting

2. Hierarchical and object oriented structure.

3 why most of the Thread functionality is specified in Object Class?

Ans: Basically for intertribal communication.

4 what is the importance of == and equals () method with respect to String object?

Ans: == is used to check whether the references are of the same object. .equals () is used to check whether the contents of the objects are the same. But with respect to strings, object reference with same content will refer to the same object.

```
String str1="Hello";
String str2="Hello";
(str1==str2) and str1.equals(str2) both will be true.
```

If you take the same example with StringBuffer, the results would be different.

```
StringBuffer str1="Hello";
StringBuffer str2="Hello";
str1.equals(str2) will be true.
str1==str2 will be false.
```

5 Is String a Wrapper Class or not?

Ans: No. String is not a Wrapper class

6 How will you find length of a String object?

Ans: Using length () method of String class.

7 How many objects are in the memory after the execution of following code segment?

Ans: String str1 = "ABC";

String str2 = "XYZ";

String str1 = str1 + str2;

There are 3 Objects.

8 what is the difference between an object and object reference?

Ans: An object is an instance of a class. Object reference is a pointer to the object. There can be many references to the same object.

9 what will trim () method of String class do?

Ans: The trim () eliminates spaces from both the ends of a string. ***

10 what is the use of java.lang.Class class?

Ans: The java.lang.Class class is used to represent the classes and interfaces that are loaded by a Java program.

11 what is the possible runtime exception thrown by substring () method?

Ans: ArrayIndexOutOfBoundsException.

12 what is the difference between String and String Buffer?

Ans: Object's of String class is immutable and object's of String Buffer class is mutable moreover String Buffer is faster in concatenation.

13 what is the use of Math class?

Ans: Math class provides methods for mathematical functions.

14 Can you instantiate Math class?

Ans: No. It cannot be instantiated. The class is final and its constructor is private. But all the methods are static, so we can use them without instantiating the Math class.

15 what will Math. abs () do?

Ans: It simply returns the absolute value of the value supplied to the method, i.e. gives you the same value. If you supply negative value it simply removes the sign.

16 what will Math. ceil () do?

Ans: This method returns always double, which is not less than the supplied value. It returns next available whole number

17 what will Math. floor () do?

Ans: This method returns always double, which is not greater than the supplied value.

18 what will Math. ax () do?

Ans: The max () method returns greater value out of the supplied values.

19 what will Math. in() do?

Ans: The min () method returns smaller value out of the supplied values.

20 what will Math. Random () do?

Ans: The random () method returns random number between 0.0 and 1.0. It always returns double.

21) java.lang package is automatically imported into all programs.

True

False

Ans : a

22) What are the interfaces defined by java.lang?

Ans: Cloneable, Comparable and Runnable.

23) What is the purpose of the Runtime class?

Ans: The purpose of the Runtime class is to provide access to the Java runtime system.

24) What is the purpose of the System class?

Ans: The purpose of the System class is to provide access to system resources.

25) Which class is extended by all other classes?

Ans: Object class is extended by all other classes.

26) Which class can be used to obtain design information about an object?

Ans: The Class class can be used to obtain information about an object's design.

27) Which method is used to calculate the absolute value of a number?

Ans : abs() method.

28) What are E and PI?

Ans : E is the base of the natural logarithm and PI is the mathematical value pi.

29) Which of the following classes is used to perform basic console I/O?

- a) System
- b) SecurityManager
- c) Math
- d) Runtime

Ans : a.

30) Which of the following are true?

- a) The Class class is the superclass of the Object class.
- b) The Object class is final.

- c) The Class class can be used to load other classes.
- d) The ClassLoader class can be used to load other classes.

Ans : c and d.

31) Which of the following methods are methods of the Math class?

- a) absolute()
- b) log()
- c) cosine()
- d) sine()

Ans : b.

32) Which of the following are true about the Error and Exception classes?

- a) Both classes extend Throwable.
- b) The Error class is final and the Exception class is not.
- c) The Exception class is final and the Error is not.
- d) Both classes implement Throwable.

Ans : a.

33) Which of the following are true?

- a) The Void class extends the Class class.
- b) The Float class extends the Double class.
- c) The System class extends the Runtime class.
- d) The Integer class extends the Number class.

Ans : d.

34) Which of the following will output -4.0

- a) System.out.println(Math.floor(-4.7));
- b) System.out.println(Math.round(-4.7));
- c) System.out.println(Math.ceil(-4.7));
- d) System.out.println(Math.Min(-4.7));

Ans : c.

35) Which of the following are valid statements

- a) public class MyCalc extends Math
- b) Math.max(s);
- c) Math.round(9.99,1);
- d) Math.mod(4,10);

e) None of the above.

Ans : e.

JAVA BY SATEESH

CHAPTER15

PACKAGES

Packages are used in Java in-order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier etc.

CALL:9010990285/7893636382

Definition:

A Package can be defined as a grouping of related types providing access protection and name space management. Note that types refer to classes, interfaces, enumerations, and annotation...etc. Enumerations and annotation types are special kinds of classes and interfaces, respectively.

Some of the existing packages in Java are:

- **java.lang** - bundles the fundamental classes
- **java.io** - classes for input , output functions are bundled in this package

Programmers can define their own packages to bundle group of classes/interfaces etc. It is a good practice to group related classes implemented by you so that a programmers can easily determine that the classes, interfaces, enumerations, annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

Creating a package:

When creating a package, you should choose a name for the package and put a **package** statement with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

The **package** statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be put into an unnamed package.

Example: Let us look at an example that creates a package called **animals**. It is common practice to use lowercased names of packages to avoid any conflicts with the names of classes, interfaces.

Note: If you put multiple types in a single source file, only one can be public, and it must have the same name as the source file. For example, you can define public class Circle in the file Circle.java, define public interface Draggable in the file Draggable.java, define public enum Day in the file Day.java, and so forth.

Put an interface in the package *animals*:

```
/* File name : Animal.java */
package animals;

interface Animal {
    public void eat();
    public void travel();
}
```

Now put an implementation in the same package *animals*:

```
package animals;

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

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

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

    public int noOfLegs(){
        return 0;
    }

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

Naming Conventions:

Package names are written in all lowercase to avoid conflict with the names of classes or interfaces.

Companies use their reversed Internet domain name to begin their package names for example, **com.example.asr** for a package named asr created by a programmer at example.com.

Name collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region or the project name after the company name (for example, com.company.region.package).

Packages in the Java language itself begin with java. or javax.

In some cases, the internet domain name may not be a valid package name. This can occur if the domain name contains a hyphen or other special character, if the package name begins with a digit or other character that is illegal to use as the beginning of a Java name, or if the package name contains a reserved Java keyword, such as "int". In this event, the suggested convention is to add an underscore. For example:

Legalizing Package Names

Domain Name	Package Name Prefix
clipart-open.org	org.clipart_open
free.fonts.int	int_.fonts.free
poetry.7days.com	com._7days.poetry

The import Keyword:

If a class wants to use another class in the same package, the package name does not need to be used. Classes in the same package find each other without any special syntax.

Example:

Here a class named Boss is added to the payroll package that already contains Employee. The Boss can then refer to the Employee class without using the payroll prefix, as demonstrated by the following Boss class.

```
package payroll;

public class Boss
{
    public void payEmployee(Employee e)
    {
        e.mailCheck();
    }
}
```

What happens if Boss is not in the payroll package? The Boss class must then use one of the following techniques for referring to a class in a different package.

- The fully qualified name of the class can be used. For example:

```
payroll. Employee
```

- The package can be imported using the import keyword and the wild card (*). For example:

```
import payroll.*;
```

- The class itself can be imported using the import keyword. For example:

```
import payroll.Employee;
```

Note: A class file can contain any number of import statements. The import statements must appear after the package statement and before the class declaration.

The Directory Structure of Packages:

Two major results occur when a class is placed in a package:

- The name of the package becomes a part of the name of the class, as we just discussed in the previous section.
- The name of the package must match the directory structure where the corresponding bytecode resides.

Here is simple way of managing your files in java:

Put the source code for a class, interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is **.java**. For example:

```
// File Name : Car.java

package vehicle;

public class Car {
    // Class implementation.
}
```

Now put the source file in a directory whose name reflects the name of the package to which the class belongs:

....\vehicle\Car.java

Now the qualified class name and pathname would be as below:

- Class name -> vehicle.Car
- Path name -> vehicle\Car.java (in windows)

In general a company uses its reversed Internet domain name for its package names. Example: A company's Internet domain name is apple.com, then all its package names would start with com.apple. Each component of the package name corresponds to a subdirectory.

Example: The company had a com.apple.computers package that contained a Dell.java source file, it would be contained in a series of subdirectories like this:

....\com\apple\computers\Dell.java

At the time of compilation, the compiler creates a different output file for each class, interface and enumeration defined in it. The base name of the output file is the name of the type, and its extension is **.class**

For example:

```
// File Name: Dell.java

package com.apple.computers;
public class Dell{

}

class Ups{



}
```

Now compile this file as follows using -d option:

```
$javac -d . Dell.java
```

This would put compiled files as follows:

```
.\com\apple\computers\Dell.class
.\com\apple\computers\Ups.class
```

You can import all the classes or interfaces defined in `\com\apple\computers\` as follows:

```
import com.apple.computers.*;
```

Like the .java source files, the compiled .class files should be in a series of directories that reflect the package name. However, the path to the .class files does not have to be the same as the path to the .java source files. You can arrange your source and class directories separately, as:

```
<path-one>\sources\com\apple\computers\Dell.java
<path-two>\classes\com\apple\computers\Dell.class
```

By doing this, it is possible to give the classes directory to other programmers without revealing your sources. You also need to manage source and class files in this manner so that the compiler and the Java Virtual Machine (JVM) can find all the types your program uses.

The full path to the classes directory, <path-two>\classes, is called the class path, and is set with the CLASSPATH system variable. Both the compiler and the JVM construct the path to your .class files by adding the package name to the class path.

Say <path-two>\classes is the class path, and the package name is com.apple.computers, then the compiler and JVM will look for .class files in <path-two>\classes\com\apple\computers.

A class path may include several paths. Multiple paths should be separated by a semicolon (Windows) or colon (Unix). By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in the class path.

Set CLASSPATH System Variable:

To display the current CLASSPATH variable, use the following commands in Windows and Unix (Bourne shell):

- In Windows -> C:\> set CLASSPATH
- In Unix -> % echo \$CLASSPATH

To delete the current contents of the CLASSPATH variable, use :

- In Windows -> C:\> set CLASSPATH=
- In Unix -> % unset CLASSPATH; export CLASSPATH

To set the CLASSPATH variable:

- In Windows -> set CLASSPATH=C:\users\jack\java\classes
- In Unix -> % CLASSPATH=/home/jack/java/classes; export CLASSPATH

CHAPTER15

MULTITHREADING

Introduction:

So far you have learned about a single thread. Lets us know about the concept of multithreading and learn the implementation of it. But before that, lets be aware from the multitasking.

Multitasking:

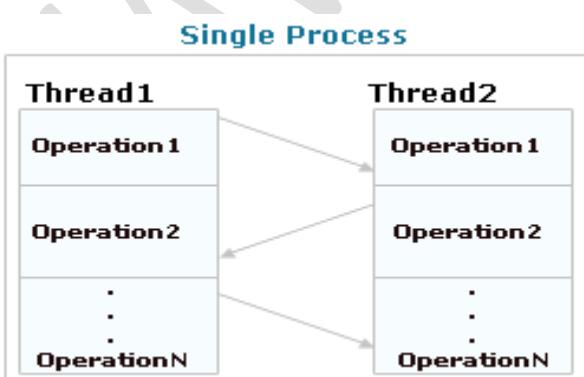
Multitasking allow to execute more than one tasks at the same time, a task being a program. In multitasking only one CPU is involved but it can switches from one program to another program so quickly that's why it gives the appearance of executing all of the programs at the same time. Multitasking allow processes (i.e. programs) to run concurrently on the program. For Example running the spreadsheet program and you are working with word processor also. Multitasking is running heavyweight processes by a single OS.

Multithreading:

Multithreading is a technique that allows a program or a process to execute many tasks concurrently (at the same time and parallel). It allows a process to run its tasks in parallel mode on a single processor system

In the multithreading concept, several multiple lightweight processes are run in a single process/task or program by a single processor. For Example, when you use a **word processor** you perform a many different tasks such as **printing, spell checking** and so on. Multithreaded software treats each process as a separate program.

In Java, the Java Virtual Machine (**JVM**) allows an application to have multiple threads of execution running concurrently. It allows a program to be more responsible to the user. When a program contains multiple threads then the CPU can switch between the two threads to execute them at the same time. For example, look at the diagram shown as:



In this diagram, two threads are being executed having more than one task. The task of each thread is switched to the task of another thread.

Advantages of multithreading over multitasking:

- Reduces the computation time.
- Improves performance of an application.
- Threads share the same address space so it saves the memory.
- Context switching between threads is usually less expensive than between processes.
- Cost of communication between threads is relatively low.

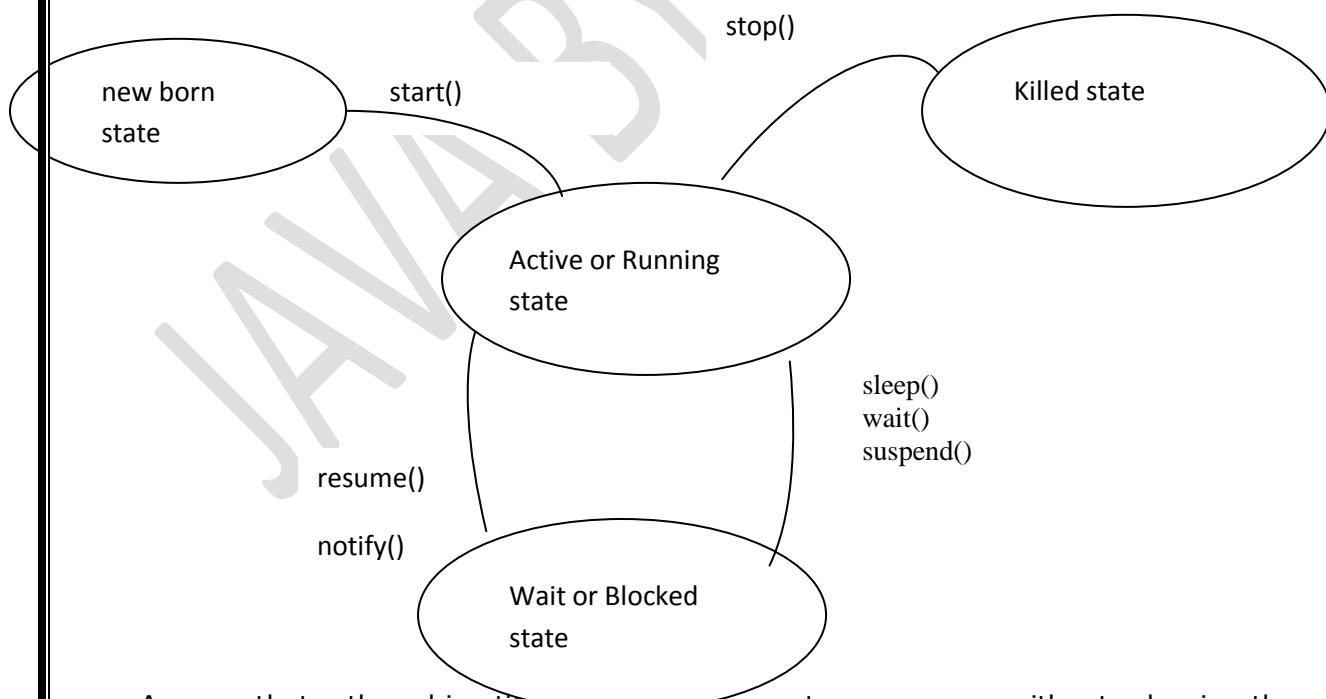
Life Cycle methods of a Thread:

The below diagram can be explained as follows:

->Once the object of a thread class is created, it will be in the "new born" state. To make a Thread class object to execute, we have to call the start () method which executes the run () method of that class. Once the thread class starts execution, it will be in "Running or Active" thread.

->An active thread, due to several reasons may be moved into "blocked" or "idle" state. If we want to make a thread for a finite amount of time, we have to call **sleep ()** method. With this method we have to specify the amount of time in milliseconds (1000 milliseconds=1 second). After the time expires, the thread will automatically move to "Running state".

-> While execution, a thread will move into different states explained as follows:



->Assume that a thread is utilizing of the system resources without releasing them. In this stage, we can block the thread for a indefinite amount of time using **suspend()** method. A suspended thread can then be moved to active state by calling **resume()** method.

->In some cases, A Thread may need to wait for an event to occur. In such a case, we can keep the thread in waiting state using **wait ()** method. A waiting thread can then be moved into active state using **notify ()** method.

->Along with the above methods, Thread class supports some more methods like join(), yield() etc.,

NOTE:

->The wait () and notify () methods of a Thread can only be called by the methods which are defined with the keywords "synchronized".

-> main () method is by default treated as a Thread by JVM. So, this method will also be executed simultaneously along with the other threads.

Like creation of a single thread, You can also create more than one thread (multithreads) in a program using class **Thread** or implementing interface **Runnable**.

Let's see an example having the implementation of the multithreads by extending **Thread** Class:

```
// Program to implement threads

public class MyThread extends Thread{

    MyThread(String s){
        super(s);
        start();
    }

    public void run(){
        for(int i=0;i<5;i++){
            System.out.println("Thread Name :"
                +Thread.currentThread().getName());
        }
    }
}
```

```
    }  
}  
  
class MultiThread1{  
  
public static void main(String args[]){  
  
    System.out.println("Thread Name :"  
  
        +Thread.currentThread().getName());  
  
    MyThread m1=new MyThread("My Thread 1");  
  
    MyThread m2=new MyThread("My Thread 2");  
  
}  
}
```

In this program, two threads are created along with the "**main**" thread. The **currentThread()** method of the **Thread** class returns a reference to the currently executing thread and the **getName()** method returns the name of the thread. The **sleep()** method pauses execution of the current thread for 1000 milliseconds(1 second) and switches to the another threads to execute it. At the time of execution of the program, both threads are registered with the **thread scheduler** and the CPU scheduler executes them one by one.

Now, lets create the same program implenting the **Runnable** interface:

Program:

```
package BASICS;  
  
class MyThread1 implements Runnable{  
  
    Thread t;  
  
    MyThread1(String s) {  
  
        t=new Thread(this,s);  
  
        t.start();  
  
    }  
}
```

```
public void run() {  
  
    for(int i=0;i<5;i++) {  
  
        System.out.println("ThreadName:"+Thread.currentThread().getName());  
  
        try {  
  
            Thread.sleep(1000);  
  
        }catch(Exception e){}  
  
    }  
  
}  
  
  
class RunnableThread1{  
  
public static void main(String args[]) {  
  
    System.out.println("ThreadName:"+Thread.currentThread().getName());  
  
    MyThread1 m1=new MyThread1("My Thread 1");  
  
    MyThread1 m2=new MyThread1("My Thread 2");  
  
}  
  
}
```

Note that, this program gives the same output as the output of the previous example. It means, you can use either class **Thread** or interface **Runnable** to implement thread in your program.

Controlling the execution of Threads:

-> Controlling thread execution implies deciding which thread is to be executed when. Generally, if not specified, execution of threads depends on the scheduling of CPU. Also, we can manually specify the sequence of execution of threads.

-> Thread Controlling involves:

CALL:9010990285/7893636382

- 1) Controlling sequence of execution of threads.
- 2) Irrespective of Quantum value, we can execute a set of statements when required i.e., we can control which part of a thread should be executed irrespective of CPU scheduling even when the time slice expires.

-> In context of Thread control, we can use the following methods:

1) Suspending a thread based on Time:

-> To suspend the execution of a thread based on time, we can use the static method of Thread class, **sleep()** which suspends a thread for a specific amount of time. This method can be used as follows:

Thread.sleep(n); where n -> milliseconds

But, sleep() method is defined to throw checked exception and so should be defined in try-catch block. The following program demonstrates this concept.

/* Program to implement sleep() method */

class A extends Thread

{

public void run()

{

try{

 sleep(3000);

 }catch(Exception e){}

}

for(int i=1;i<=10;i++)

{

 System.out.println("Inside A " + i);

}

}// end of thread A

class B extends Thread

CALL:9010990285/7893636382

```
{  
public void run()  
{  
try{  
    sleep(1500);  
}catch(Exception e){}  
  
}  
for(int i=1;i<=10;i++)  
{  
System.out.println("Inside B " + i);  
}  
} // end of Thread B  
  
class C extends Thread  
{  
public void run()  
{  
for(int i=1;i<=10;i++)  
System.out.println("Inside C " + i);  
}  
}// End of Thread C  
  
class ThreadDemo2  
{  
public static void main(String args[])  
{  
A a1=new A();  
}
```

```
B b1=new B();  
C c1=new C();  
  
a1.start();  
  
b1.start();  
  
c1.start();  
  
}  
  
} // end of ThreadDemo2
```

-> In any case, when you execute the above program, you observe that Thread C starts execution first, then Thread B and finally Thread A starts its execution.

2) Suspending a thread until the execution of another thread:

-> For this purpose, we can use a non-static method called **join()** which makes a thread wait until another thread completes its execution. The following program demonstrates this concept.

/* Program to implement join method */

```
class A extends Thread  
  
{  
  
    int sum;  
  
    public void run()  
    {  
  
        for(int i=1;i<=10;i++)  
  
            sum+=i;  
  
        System.out.println("Sum in A = " + sum);  
  
        try{  
  
            Thread.sleep(100);  
  
        }  
  
        catch(Exception e){System.out.println(e);}  
    }  
}
```

```
        System.out.println("End of Thread A");  
    }  
}// end of thread A
```

```
class B extends Thread
```

```
{  
  
    public void run()  
  
    {  
  
        for(int j=1;j<=10;j++)  
  
            System.out.println("j = " + j);  
  
    }  
} // end of thread B
```

```
class C extends Thread
```

```
{  
  
    A a1;  
  
    C(A a1)  
  
    {  
  
        this.a1 = a1;  
  
    }  
  
    public void run()  
  
    {  
  
        for(int i=1;i<=10;i++)  
  
    }  
}
```

```
        System.out.println("j = " + j);  
  
        if(i == 5)  
  
    {
```

```
a1.join();

System.out.println("Sum = " + a1.sum);

} // if

} // for

} run

}// end class

class JoinDemo

{

public static void main(String args[])

{

A a1 = new A();

B b1 = new B();

C c1 = new C(a1);

a1.start();

b1.start();

c1.start();

} // main

} // class
```

3) Suspending the execution of a thread unconditionally:

-> To suspend an executing thread without any condition, we can use a non-static method of Thread class called **suspend()** and a suspended thread can be restarted using **resume()** method.

-> But, the above two functions are deprecated because they are proven to create deadlocks.

-> For the same purpose, we can use **wait()** and **notify()** methods which does not lead to deadlock. These methods belong to Object class and not thread class. These methods are only used in the methods which are defined as synchronized.

Thread Priorities:

In Java, thread scheduler can use the thread **priorities** in the form of **integer value** to each of its thread to determine the execution schedule of threads. Thread gets the **ready-to-run** state according to their priorities. The **thread scheduler** provides the CPU time to thread of highest priority during ready-to-run state.

Priorities are integer values from 1 (lowest priority given by the constant **(Thread.MIN_PRIORITY)**) to 10 (highest priority given by the constant **(Thread.MAX_PRIORITY)**). The default priority is 5 (**Thread.NORM_PRIORITY**).

The methods that are used to set the priority of thread shown as:

Method	Description
setPriority()	This is method is used to set the priority of thread.
getPriority()	This method is used to get the priority of thread.

When a Java thread is created, it inherits its priority from the thread that created it. At any given time, when multiple threads are ready to be executed, the runtime system chooses the runnable thread with the highest priority for execution. In Java runtime system, **preemptive scheduling** algorithm is applied. If at the execution time a thread with a higher priority and all other threads are runnable then the runtime system chooses the new **higher priority** thread for execution. On the other hand, if two threads of the same priority are waiting to be executed by the CPU then the **round-robin** algorithm is applied in which the scheduler chooses one of them to run according to their round of **time-slice**.

Thread Scheduler:

In the implementation of threading scheduler usually applies one of the two following strategies:

- **Preemptive scheduling:** If the new thread has a higher priority, then current running thread leaves the runnable state and higher priority thread enter to the runnable state.

- **Time-Sliced (Round-Robin) Scheduling:** A running thread is allowed to execute for the fixed time, after completion the time, current thread indicates to the another thread to enter it in the runnable state.

You can also set a thread's priority at any time after its creation using the **set Priority** method. Let's see, how to set and get the priority of a thread.

Example:

```
class MyThread1 extends Thread{  
    MyThread1(String s){  
        super(s);  
        start();  
    }  
    public void run(){  
        for(int i=0;i<3;i++){  
            Thread cur=Thread.currentThread();  
            cur.setPriority(Thread.MIN_PRIORITY);  
            int p=cur.getPriority();  
            System.out.println("Thread Name :" + Thread.currentThread().getName());  
            System.out.println("Thread Priority :" + cur);  
        }  
    }  
}  
  
class MyThread2 extends Thread{  
    MyThread2(String s){  
        super(s);  
        start();  
    }  
    public void run(){  
        for(int i=0;i<3;i++){  
            Thread cur=Thread.currentThread();  
            cur.setPriority(Thread.MAX_PRIORITY);  
            int p=cur.getPriority();  
            System.out.println("Thread Name :" + Thread.currentThread().getName());  
            System.out.println("Thread Priority :" + cur);  
        }  
    }  
}
```

```

}

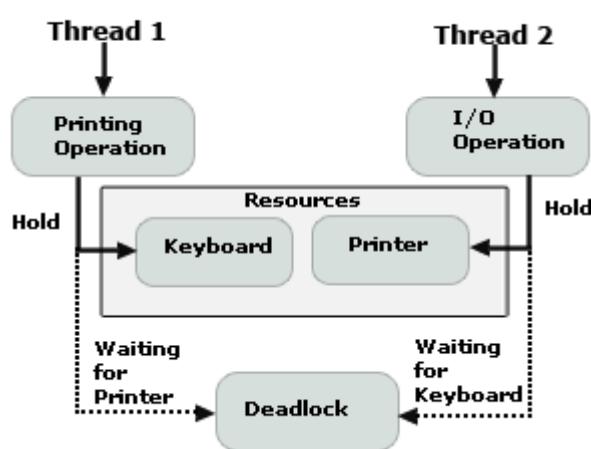
public class ThreadPriority{
    public static void main(String args[]){
        MyThread1 m1=new MyThread1("My Thread 1");
        MyThread2 m2=new MyThread2("My Thread 2");
    }
}

```

In this program two threads are created. We have set up maximum priority for the first thread "MyThread2" and minimum priority for the first thread "MyThread1" i.e. the after executing the program, the first thread is executed only once and the second thread "MyThread2" started to run until either it gets end or another thread of the equal priority gets ready to run state.

Deadlock:

A situation where a thread is waiting for an object lock that holds by second thread and this second thread is waiting for an object lock that holds by first thread, this situation is known as **Deadlock**. Let's see a situation in the diagram shown below where the deadlock condition is occurred:



In this diagram two threads having the **Printing** & **I/O** operations respectively at a time. But **Thread1** need to **printer** that is hold up by the **Thread2**, likewise **Thread2** need the **keyboard** that is hold up by the Thread1. In this situation the CPU becomes ideal and the deadlock condition occurs because no one thread is executed until the holdup resources are free.

Example:

The following program demonstrates the deadlock situation:

```
package BASICS;

public class DeadDemo{

    public static void main(String args[]){

        String s1="Dead";

        String s2="Lock";

        MyThread1 m=new MyThread1(s1,s2);

        MyThread2 m1=new MyThread2(s1,s2);

    }

}

class MyThread1 extends Thread{

    String s1;

    String s2;

    MyThread1(String s1, String s2){

        this.s1=s1;

        this.s2=s2;

        start();

    }

    public void run(){

        while(true){

            synchronized(s1){

                synchronized(s2){

                    System.out.println(s1+s2);

                }

            }

        }

    }

}
```

```
        }

    }

}

}

class MyThread2 extends Thread{

    String s1;

    String s2;

    MyThread2(String s1,String s2){

        this.s1=s1;

        this.s2=s2;

        start();

    }

    public void run(){

        while(true){

            synchronized(s2){

                synchronized(s1){

                    System.out.println(s2+s1);

                }

            }

        }

    }

}
```

Synchronization in Java:

-> Synchronization is the concept needed when multiple threads need to collaborate with one other. Mutual Exclusion is the major concept of Synchronization which means that when a thread is under execution, the remaining threads must go into waiting state.

-> Producer-Consumer Problem is the best example where synchronization is required. Assume that in an application, there are two threads defined Producer Thread and Consumer Thread. Producer is the thread which produces items and places them in a common buffer. Consumer is the thread which collects the items from the buffer and consumes them. Printing a file is the best example for P-C problem where disk controller is the producer and printer controller is the consumer.

-> In this application, while the producer is running, consumer has to wait and vice-versa. Here, we need to define the produce() and consume() methods as "synchronized" because to implement wait() and notify() methods of Thread class, we have to define them in synchronized methods only.

-> The following program demonstrates "P-C" problem with synchronization:

//using synchronized keyword

```
class Common
{
    int i;
    boolean flag=true;

    /* If flag is true, Producer will get executed and makes consumer to be in waiting state. After Completion, producer makes Consumer Thread to get executed by making flag as false. Stated simply, if flag is true, Consumer will wait and producer will wait if flag is false.*/
    synchronized void produce(int j)
    {
        if(flag)
        {
            i=j;
            System.out.println("Produced Item = "+ i);
            flag=false;
            notify();//making Consumer to get Executed
        }
    }
}
```

```
try{  
    wait();}  
catch(Exception e)  
{ System.out.println(e);}  
  
}  
  
synchronized int consume()  
{  
try{  
    if(flag)  
    wait();  
}  
catch(Exception e)  
{  
    System.out.println(e);  
}  
flag=true;  
notify();  
return i;  
}  
}  
}  
  
class Producer extends Thread  
{  
Common c1;  
Producer(Common c1)  
{ this.c1=c1; }  
public void run()  
{
```

```
int i=0;  
while(true)  
{  
    i++;  
    c1.produce(i);  
    try{sleep(1000);}catch(Exception e)  
    {System.out.println(e);}  
}  
}  
}  
  
class Consumer extends Thread  
{  
    Common c1;  
    Consumer(Common c1)  
    { this.c1=c1; }  
    public void run()  
    {  
        while(true)  
        {  
            int j=c1.consume();  
            System.out.println("Consumed item= " + j);  
            try{sleep(1000);}catch(Exception e)  
            {System.out.println(e);}  
        }  
    }  
}
```

```
}
```

```
class SynDemo2
```

```
{
```

```
public static void main(String args[])
{
    Common c1=new Common();
    Producer p=new Producer(c1);
    Consumer c=new Consumer(c1);
    p.start();
    c.start();
}
```

Runnable interface:

-> In multithreading, we can create a thread class either by extending `java.lang.Thread` class or by implementing `Runnable` interface.

->The class to be created as a Thread, if it not extending any class, we have to define the class to extend "`java.lang.Thread`" class. If that class is already extending some class, then it could not extend `java.lang.Thread` class because this creates Multiple Inheritance. So, for this class , it should implement `Runnable` interface.

->The following program demonstrates the use of `Runnable` interface.

// Program to implement Runnable interface

```
class A
{
    void fun1()
    {
        System.out.println("Inside fun1");
    }
}
```

```
}

void fun2()

{

System.out.println("Inside fun2");

}

}

class B extends A implements Runnable

{

public void run()

{

for(int i=0;i<10;i++)

System.out.println("Inside B " + i);

}

}

class RunnableDemo

{

public static void main(String args[])

{

B b1= new B();

b1.fun1();

b1.fun2();

Thread t=new Thread(b1);

t.start();

for(int i=0;i<5;i++)

System.out.println("Inside main");

}
```

```

    }
}

}

```

-> In the above program, because Runnable interface does not have start() method, the object of Thread B is passed as argument to the constructor of Thread class and through the Thread class object, we are calling start() method [t.start()].

Inter-Thread Communication:

Java provides a very efficient way through which multiple-threads can communicate with each-other. This way reduces the CPU's idle time i.e. A process where, a thread is paused running in its critical region and another thread is allowed to enter (or lock) in the same critical section to be executed. This technique is known as **Inter thread communication** which is implemented by some methods. These methods are defined in "**java.lang**" package and can only be called within synchronized code shown as:

Method	Description
wait()	It indicates the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls method notify() or notifyAll() .
notify()	It wakes up the first thread that called wait() on the same object.
notifyAll()	Wakes up (Unlock) all the threads that called wait() on the same object. The highest priority thread will run first.

All these methods must be called within a try-catch block.

Let's see an example implementing these methods:

```

class Shared {
    int num=0;
    boolean value = false;

    synchronized int get() {
        if (value==false)

```

```
try {
    wait();
}
catch (InterruptedException e) {
    System.out.println("InterruptedException caught");
}
System.out.println("consume: " + num);
value=false;
notify();
return num;
}

synchronized void put(int num) {
    if (value==true)
        try {
            wait();
        }
    catch (InterruptedException e) {
        System.out.println("InterruptedException caught");
    }
    this.num=num;
    System.out.println("Produce: " + num);
    value=false;
    notify();
}
}

class Producer extends Thread {
Shared s;

Producer(Shared s) {
    this.s=s;
    this.start();
}

public void run() {
    int i=0;
```

```
s.put(++i);
}

}

class Consumer extends Thread{
    Shared s;

    Consumer(Shared s) {
        this.s=s;
        this.start();
    }

    public void run() {
        s.get();
    }
}

public class InterThread{
    public static void main(String[] args)
    {
        Shared s=new Shared();
        new Producer(s);
        new Consumer(s);
    }
}
```

Output of the Program:

```
C:\nisha>javac
InterThread.java

C:\nisha>java
InterThread
Produce:          1
consume: 1
```

In this program, two threads "**Producer**" and "**Consumer**" share the **synchronized** methods of the class "**Shared**". At time of program execution, the "**put()**" method is invoked through the "**Producer**" class which increments the variable "**num**" by **1**. After producing 1 by the producer, the method "**get()**" is invoked by through the "**Consumer**" class which retrieves the produced number and returns it to the output. Thus the Consumer can't retrieve the number without producing of it.

Another program demonstrates the uses of wait() & notify() methods:

```
public class DemoWait extends Thread{
    int val=20;
    public static void main(String args[]) {
        DemoWait d=new DemoWait();
        d.start();
        new Demo1(d);
    }
    public void run(){
        try {
            synchronized(this){
                wait();
                System.out.println("value is :" +val);
            }
        }catch(Exception e){}
    }

    public void valchange(int val){
        this.val=val;
        try {
            synchronized(this) {
                notifyAll();
            }
        }catch(Exception e){}
    }
}

class Demo1 extends Thread{
    DemoWait d;
    Demo1(DemoWait d) {
```

```
this.d=d;
start();
}
public void run(){
try{
System.out.println("Demo1 value is"+d.val);
d.valchange(40);
}catch(Exception e){}
}
}
```

Output of the program is:

```
C:\j2se6\thread>javac
DemoWait.java

C:\j2se6\thread>java
DemoWait
Demo1    value    is20
value      is      :40

C:\j2se6\thread>
```

Daemon Threads:

In Java, any thread can be a Daemon thread. Daemon threads are like a **service providers** for other threads or objects running in the same process as the daemon thread. Daemon threads are used for background supporting tasks and are only needed while normal threads are executing. If normal threads are not running and remaining threads are daemon threads then the interpreter exits.

setDaemon(true/false): This method is used to specify that a thread is daemon thread.
public boolean isDaemon(): This method is used to determine the thread is daemon thread or not.

The following program demonstrates the **Daemon Thread:**

```
public class DaemonThread extends Thread {  
    public void run() {  
        System.out.println("Entering run method");  
  
        try {  
            System.out.println("In run Method: currentThread() is"  
                + Thread.currentThread());  
  
            while (true) {  
                try {  
                    Thread.sleep(500);  
                } catch (InterruptedException x) {  
                }  
  
                System.out.println("In run method: woke up again");  
            }  
        } finally {  
            System.out.println("Leaving run Method");  
        }  
    }  
  
    public static void main(String[] args) {  
        System.out.println("Entering main Method");  
  
        DaemonThread t = new DaemonThread();  
        t.setDaemon(true);  
        t.start();  
  
        try {  
            Thread.sleep(3000);  
        } catch (InterruptedException x) {  
        }  
  
        System.out.println("Leaving main method");  
    }  
}
```

Output of this program is:

CALL:9010990285/7893636382

```
C:\j2se6\thread>javac  
DaemonThread.java  
  
C:\j2se6\thread>java DaemonThread  
Entering main Method  
Entering run method  
In run Method: currentThread()  
isThread[Thread-0,5,main]  
In run method: woke up again  
Leaving main method  
  
C:\j2se6\thread>
```

INTERVIEW QUESTIONS ON MULTITHREADING:

Describe synchronization in respect to multithreading.

Ans: With respect to multithreading, synchronization is the capability to control the access of multiple threads to shared resources. Without synchronization, it is possible for one thread to modify a shared variable while another thread is in the process of using or updating same shared variable. This usually leads to significant errors.

Explain different way of using thread?

Ans: The thread could be implemented by using runnable interface or by inheriting from the Thread class. The former is more advantageous, 'cause when you are going for multiple inheritance..the only interface can help.

What are the two types of multitasking?

Ans :

1. process-based
2. Thread-based

What are the two ways to create the thread?

Ans :

CALL:9010990285/7893636382

- 1.by implementing Runnable
- 2.by extending Thread

What is the signature of the constructor of a thread class?

Ans : Thread(Runnable threadob, String threadName)

What are all the methods available in the Runnable Interface?

Ans : run()

What is the data type for the method isAlive() and this method is available in which class?

Ans : boolean, Thread

What are all the methods available in the Thread class?

Ans :

- 1.isAlive()
- 2.join()
- 3.resume()
- 4.suspend()
- 5.stop()
- 6.start()
- 7.sleep()
- 8.destroy()

What are all the methods used for Inter Thread communication and what is the class in which these methods are defined?

Ans :

1. wait(), notify() & notifyall()
2. Object class

What is the mechanism defined by java for the Resources to be used by only one Thread at a time?

Ans : Synchronisation

What is the procedure to own the monitor by many threads?

Ans : not possible

What is the unit for 1000 in the below statement?

CALL:9010990285/7893636382

ob.sleep(1000)

Ans : long milliseconds

What is the data type for the parameter of the sleep() method?

Ans : long

What are all the values for the following level?

max-priority

min-priority

normal-priority

Ans : 10,1,5

What is the method available for setting the priority?

Ans : setPriority()

What is the default thread at the time of starting the program?

Ans : main thread

The word synchronized can be used with only a method.

True/ False

Ans : False

Which priority Thread can prompt the lower primary Thread?

Ans : Higher Priority

How many threads at a time can access a monitor?

Ans : one

What are all the four states associated in the thread?

Ans : 1. new 2. runnable 3. blocked 4. dead

The suspend() method is used to terminate a thread?

True /False

Ans : False

The run() method should necessarily exists in classes created as subclass of thread?

True /False

Ans : True

When two threads are waiting on each other and can't proceed the programme is said to be in a deadlock?

True/False

Ans : True

Which method waits for the thread to die ?

Ans : join() method

Which of the following is true?

1) wait(),notify(),notifyall() are defined as final & can be called only from within a synchronized method

2) Among wait(),notify(),notifyall() the wait() method only throws IOException

3) wait(),notify(),notifyall() & sleep() are methods of object class

- 1
- 2
- 3
- 1 & 2
- 1,2 & 3

Ans : D

Garbage collector thread belongs to which priority?

Ans : low-priority

What is meant by timeslicing or time sharing?

Ans : Timeslicing is the method of allocating CPU time to individual threads in a priority schedule.

What is meant by daemon thread? In java runtime, what is its role?

Ans : Daemon thread is a low priority thread which runs intermittently in the background doing the garbage collection operation for the java runtime system.

CHAPTER18

AWT

Introduction to Graphical User Interfaces (GUI)

The Java programming language provides a class library called the Abstract Window Toolkit (AWT) that contains a number of common graphical widgets. You can add these widgets to your display area and position them with a layout manager.

AWT Basics:

All graphical user interface objects stem from a common superclass, Component. To create a Graphical User Interface (GUI), you add components to a Container object. Because a Container is also a Component, containers may be nested arbitrarily. Most often, you will use a Panel when creating nested GUIs.

Each AWT component uses native code to display itself on your screen. When you run a Java application under Microsoft Windows, buttons are really Microsoft Windows buttons. When you run the same application on a Macintosh, buttons are really Macintosh buttons. When you run on a UNIX machine that uses Motif, buttons are really Motif buttons.

Applications versus Applets:

Recall that an Applet is a Java program that runs in a web page, while an application is one that runs from the command line. An Applet is a Panel that is automatically inserted into a web page. The browser displaying the web page instantiates and adds the Applet to the proper part of the web page. The browser tells the Applet when to create its GUI (by calling the init() method of Applet) and when to start() and stop() any special processing.

Applications run from a command prompt. When you execute an application from the command prompt, the interpreter starts by calling the application's main() method.

Basic GUI Logic:

There are three steps you take to create any GUI application or applet:

Compose your GUI by adding components to Container objects.

Setup event handlers to respond to user interaction with the GUI.

Display the GUI (automatically done for applets, you must explicitly do this for applications).

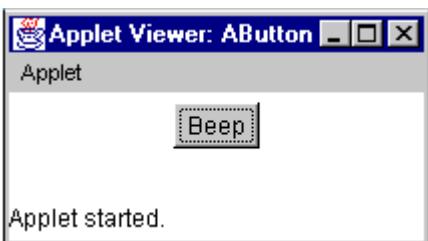
When you display an AWT GUI, the interpreter starts a new thread to watch for user interaction with the GUI. This new thread sits and waits until a user presses a key, clicks or moves the mouse, or any other system-level event that affects the GUI. When it receives

such an event, it calls one of the event handlers you set up for the GUI. Note that the event handler code is executed within the thread that watches the GUI! (This is an important point that will be revisited later when events are reviewed.)

Because this extra thread exists, your main method can simply end after it displays the GUI. This makes GUI code very simple to write in AWT. Compose the GUI, setup event handlers, then display.

A Simple Example

The following simple example shows some GUI code. This example creates an Applet that contains just an Applet.



```
import java.awt.Button;  
  
import java.applet.Applet;  
  
  
public class AButton extends Applet {  
  
    public void init() {  
  
        // STEP 1: Compose the GUI  
  
        Button beepButton = new Button("Beep");  
  
        add(beepButton);  
  
  
        // STEP 2: Setup Event handlers  
  
        beepButton.addActionListener(new Beeper());
```

```
// STEP 3: Display the GUI (automatic -- this is an applet)  
}  
}
```

In step 2 from above, event handling is set up by adding an instance of a listener class to the button. When the button is pressed, a certain method in the listener class is called. In this example, the listener class implements ActionListener (because Button requires it). When the button is pressed, the button calls the actionPerformed() method of the listener class. Event handling is discussed in detail later in this module.

Suppose you want to produce a "beep" sound when the button is pressed. You can define your event handler as follows:

```
import java.awt.event.ActionListener;  
  
import java.awt.event.ActionEvent;  
  
import java.awt.Component;  
  
  
public class Beeper implements ActionListener {  
  
    public void actionPerformed(ActionEvent event) {  
  
        Component c = (Component)event.getSource();  
  
        c.getToolkit().beep();  
  
    }  
}
```

When actionPerformed() is called, it produces a beep sound on the computer (assuming your system is sound capable).

To try this applet, create a simple HTML page as follows.

```
<html>
```

CALL:9010990285/7893636382

```
<applet code=AButton.class width=100 height=100>  
</applet>  
</html>
```

Then test the HTML page by running appletviewer or by loading the HTML file in a browser that supports the Java Runtime Environment (JRE). Note that in this case, the browser must support at least version 1.1 of the JRE, as the example uses the event handling capabilities introduced with that release.

AWT Components:

All AWT components extend class Component. Think of Component as the "root of all evil" for AWT. Having this single class is rather useful, as the library designers can put a lot of common code into it.

Next, examine each of the AWT components below. Most, but not all, directly extend Component. You've used most of the components should be familiar to you.

Buttons:

A Button has a single line label and may be "pushed" with a mouse click.



```
import java.awt.*;  
  
import java.applet.Applet;  
  
  
public class ButtonTest extends Applet {  
  
    public void init() {  
  
        Button button = new Button("OK");  
    }  
}
```

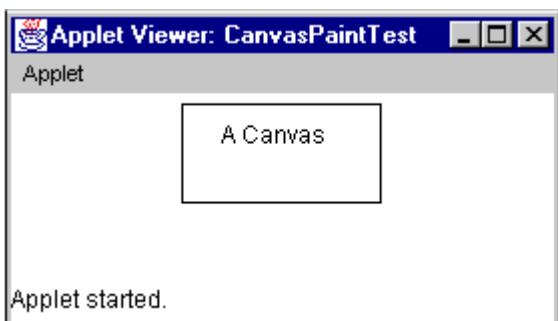
```
    add(button);  
}  
}
```

Note that in the above example there is no event handling added; pressing the button will not do anything.

The AWT button has no direct support for images as labels.

Canvas:

A Canvas is a graphical component representing a region where you can draw things such as rectangles, circles, and text strings. The name comes from a painter's canvas. You subclass Canvas to override its default paint() method to define your own components.



You can subclass Canvas to provide a custom graphic in an applet.

```
import java.awt.Canvas;  
  
import java.awt.Graphics;  
  
class DrawingRegion extends Canvas {  
  
    public DrawingRegion() {  
  
        setSize(100, 50);  
    }
```

```
}

public void paint(Graphics g) {
    g.drawRect(0, 0, 99, 49); // draw border
    g.drawString("A Canvas", 20, 20);
}

}
```

Then you use it like any other component, adding it to a parent container, for example in an Applet subclass.

```
import java.applet.Applet;

public class CanvasPaintTest extends Applet {

    public void init() {
        DrawingRegion region = new DrawingRegion();
        add(region);
    }
}
```

The Canvas class is frequently extended to create new component types, for example image buttons. However, starting with the JRE 1.1, you can now directly subclass Component directly to create lightweight, transparent widgets.

Checkbox:

A Checkbox is a label with a small pushbutton. The state of a Checkbox is either true (button is checked) or false (button not checked). The default initial state is false. Clicking a Checkbox toggles its state. For example:



```
import java.awt.*;  
  
import java.applet.Applet;  
  
  
public class CheckboxSimpleTest extends Applet {  
  
    public void init() {  
  
        Checkbox m = new Checkbox("Allow Mixed Case");  
  
        add(m);  
  
    }  
  
}
```

To set a Checkbox initially true use an alternate constructor:

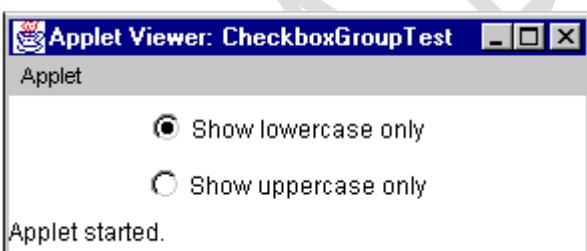


```
import java.awt.*;  
  
import java.applet.Applet;  
  
  
public class CheckboxSimpleTest2 extends Applet {  
  
    public void init() {  
  
        Checkbox m = new Checkbox("Label", true);  
  
        add(m);  
  
    }  
  
}
```

CheckboxGroup:

A CheckboxGroup is used to control the behavior of a group of Checkbox objects (each of which has a true or false state). Exactly one of the Checkbox objects is allowed to be true at one time. Checkbox objects controlled with a CheckboxGroup are usually referred to as "radio buttons".

The following example illustrates the basic idea behind radio buttons.

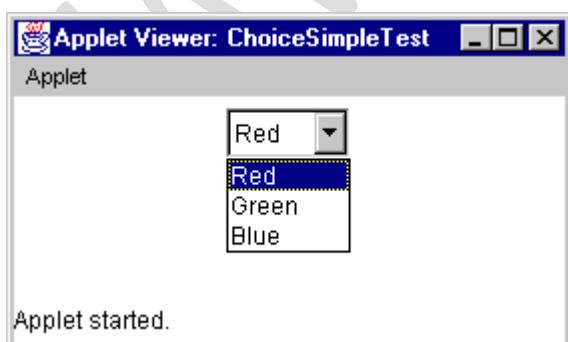


```
import java.awt.*;  
  
import java.applet.Applet;  
  
  
public class CheckboxGroupTest extends Applet {
```

```
public void init() {  
    // create button controller  
  
    CheckboxGroup cbg = new CheckboxGroup();  
  
  
    Checkbox cb1 =  
        new Checkbox("Show lowercase only", cbg, true);  
  
    Checkbox cb2 =  
        new Checkbox("Show uppercase only", cbg, false);  
  
  
    add(cb1);  
    add(cb2);  
}  
}
```

Choice:

Choice objects are drop-down lists. The visible label of the Choice object is the currently selected entry of the Choice.

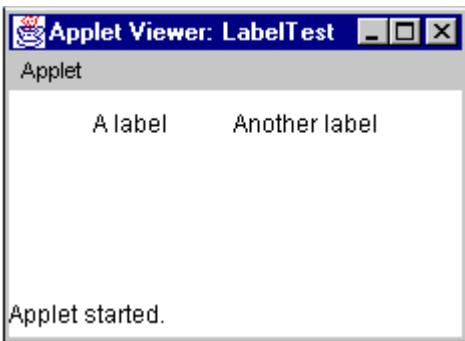


```
import java.awt.*;  
  
import java.applet.Applet;  
  
  
public class ChoiceSimpleTest extends Applet {  
  
    public void init() {  
  
        Choice rgb = new Choice();  
  
  
        rgb.add("Red");  
  
        rgb.add("Green");  
  
        rgb.add("Blue");  
  
  
        add(rgb);  
  
    }  
  
}
```

The first item added is the initial selection.

Label:

A Label is a displayed Label object. It is usually used to help indicate what other parts of the GUI do, such as the purpose of a neighboring text field.



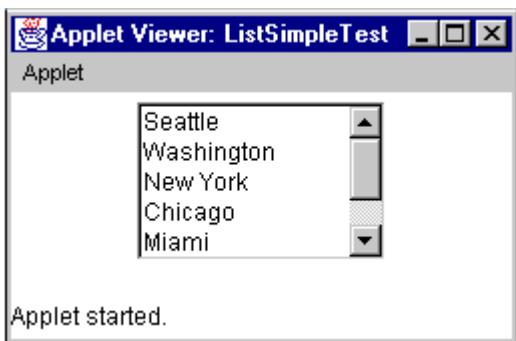
```
import java.awt.*;  
  
import java.applet.Applet;  
  
  
public class LabelTest extends Applet {  
  
    public void init() {  
  
        add(new Label("A label"));  
  
        // right justify next label  
  
        add(new Label("Another label", Label.RIGHT));  
  
    }  
  
}
```

Like the Button component, a Label is restricted to a single line of text.

List:

A List is a scrolling list box that allows you to select one or more items.

Multiple selections may be used by passing true as the second argument to the constructor.



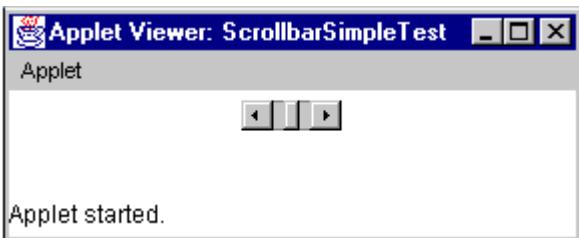
```
import java.awt.*;  
  
import java.applet.Applet;  
  
  
public class ListSimpleTest extends Applet {  
  
    public void init() {  
  
        List list = new List(5, false);  
  
        list.add("Seattle");  
  
        list.add("Washington");  
  
        list.add("New York");  
  
        list.add("Chicago");  
  
        list.add("Miami");  
  
        list.add("San Jose");  
  
        list.add("Denver");  
  
  
        add(list);  
  
    }  
  
}
```

The constructor may contain a preferred number of lines to display. The current LayoutManager may choose to respect or ignore this request.

CALL:9010990285/7893636382

Scrollbar:

A Scrollbar is a "slider" widget with characteristics specified by integer values that are set during Scrollbar construction. Both horizontal and vertical sliders are available.



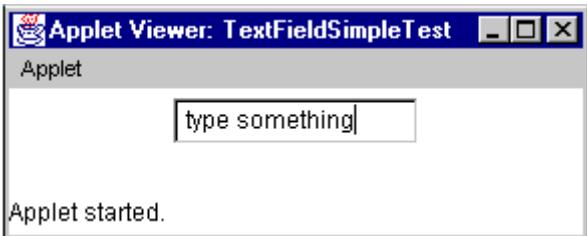
```
import java.awt.*;  
  
import java.applet.Applet;  
  
  
// A simple example that makes a Scrollbar appear  
  
public class ScrollbarSimpleTest extends Applet {  
  
    public void init() {  
  
        Scrollbar sb =  
  
            new Scrollbar(Scrollbar.HORIZONTAL,  
  
                          0, // initial value is 0  
  
                          5, // width of slider  
  
                          -100, 105); // range -100 to 100  
  
        add(sb);  
  
    }  
  
}
```

The maximum value of the Scrollbar is determined by subtracting the Scrollbar width from the maximum setting (last parameter).

CALL:9010990285/7893636382

TextField:

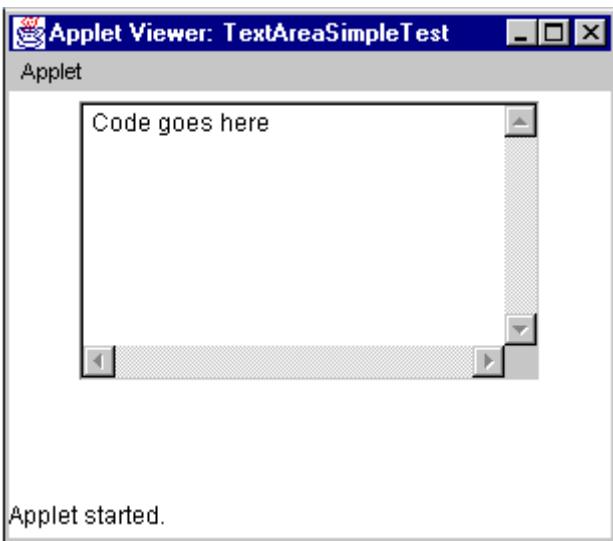
A TextField is a scrollable text display object with one row of characters. The preferred width of the field may be specified during construction and an initial string may be specified.



```
import java.awt.*;  
  
import java.applet.Applet;  
  
  
public class TextFieldSimpleTest extends Applet {  
  
    public void init() {  
  
        TextField f1 =  
  
            new TextField("type something");  
  
        add(f1);  
  
    }  
  
}
```

TextArea:

A TextArea is a multi-row text field that displays a single string of characters, where newline ('\n' or '\n\r' or '\r', depending on platform) ends each row. The width and height of the field is set at construction, but the text can be scrolled up/down and left/right.



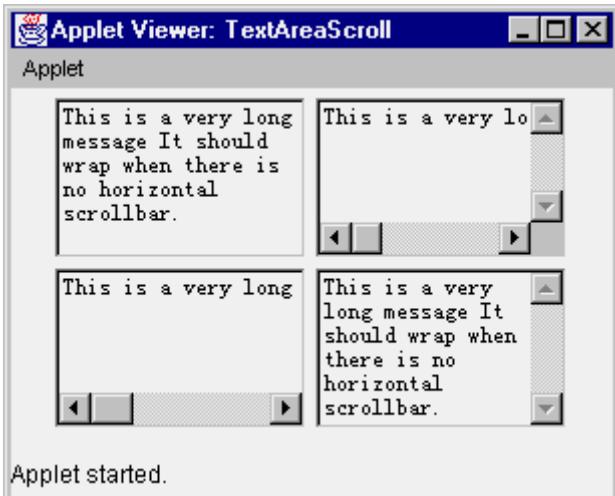
```
import java.awt.*;  
  
import java.applet.Applet;  
  
  
public class TextAreaSimpleTest extends Applet {  
  
    TextArea disp;  
  
    public void init() {  
  
        disp = new TextArea("Code goes here", 10, 30);  
  
        add(disp);  
  
    }  
  
}
```

There is no way, for example, to put the cursor at beginning of row five, only to put the cursor at single dimension position 50.

There is a four-argument constructor that accepts a fourth parameter of a scrollbar policy.

The different settings are the class constants: SCROLLBARS_BOTH, SCROLLBARS_HORIZONTAL_ONLY, SCROLLBARS_NONE, and

SCROLLBARS_VERTICAL_ONLY. When the horizontal (bottom) scrollbar is not present, the text will wrap.



```
import java.awt.*;
import java.applet.Applet;

public class TextAreaScroll extends Applet {

    String s =
        "This is a very long message " +
        "It should wrap when there is " +
        "no horizontal scrollbar.";

    public void init() {
        add(new TextArea (s, 4, 15,
                         TextArea.SCROLLBARS_NONE));
        add(new TextArea (s, 4, 15,
                         TextArea.SCROLLBARS_BOTH));
        add(new TextArea (s, 4, 15,
```

```
TextArea.SCROLLBARS_HORIZONTAL_ONLY));  
add(new TextArea (s, 4, 15,  
TextArea.SCROLLBARS_VERTICAL_ONLY));  
}  
}
```

Common Component Methods:

All AWT components share the 100-plus methods inherited from the Component class. Some of the most useful and commonly-used methods are listed below:

getSize() - Gets current size of component, as a Dimension.


```
Dimension d = someComponent.getSize();  
int height = d.height;  
int width = d.width;
```

Note: With the Java 2 Platform, you can directly access the width and height using the getWidth() and getHeight() methods. This is more efficient, as the component doesn't need to create a new Dimension object. For example:

```
int height = someComponent.getHeight();  
// Java 2 Platform only!  
int width = someComponent.getWidth();  
// Java 2 Platform only!
```

If you're using the Java 2 platform, you should only use getSize() if you really need a Dimension object!

getLocation() - Gets position of component, relative to containing component, as a Point.

```
Point p = someComponent.getLocation();
int x = p.x;
int y = p.y;
```

Note: With the Java 2 Platform, you can directly access the x and y parts of the location using `getX()` and `getY()`. This is more efficient, as the component doesn't have to create a new Point object. For example:

```
int x = someComponent.getX();
// Java 2 Platform only!
int y = someComponent.getY();
// Java 2 Platform only!
```

If you're using the Java 2 platform, you should only use `getLocation()` if you really need a Point object!

`getLocationOnScreen()` - Gets the position of the component relative to the upper-left corner of the computer screen, as a Point.

```
Point p = someComponent.getLocationOnScreen();
```

```
int x = p.x;
int y = p.y;
```

`getBounds()` - Gets current bounding Rectangle of component.

```
Rectangle r = someComponent.getBounds();
int height = r.height;
int width = r.width;
```

```
int x = r.x;  
int y = r.y;
```

This is like a combination of calling `getLocation()` and `getSize()`. Note: If you're using the Java 2 Platform and don't really need a `Rectangle` object, you should use `getX()`, `getY()`, `getWidth()`, and `getHeight()` instead.

`setEnabled(boolean)` - Toggles the state of the component. If set to true, the component will react to user input and appear normal. If set to false, the component will ignore user interaction, and usually appear ghosted or grayed-out.

`setVisible(boolean)` - Toggles the visibility state of the component. If set to true, the component will appear on the screen if it is contained in a visible container. If false, the component will not appear on the screen. Note that if a component is marked as not visible, any layout manager that is responsible for that component will usually proceed with the layout algorithm as though the component were not in the parent container! This means that making a component invisible will not simply make it disappear while reserving its space in the GUI. Making the component invisible will cause the layout of its sibling components to readjust!

`setBackground(Color)/setForeground(Color)` - Changes component background/foreground colors.

`setFont(Font)` - Changes font of text within a component.

Containers:

A Container is a Component, so may be nested. Class Panel is the most commonly-used Panel and can be extended to partition GUIs. Class Applet is a specialized Panel for running programs within a browser.

Common Container Methods

Besides the 100-plus methods inherited from the `Component` class, all Container subclasses inherit the behavior of about 50 common methods of `Container` (most of which just override a method of `Component`). While the most common method of `Container` used `add()`, has already been briefly discussed, if you need to access the list of components within a

container, you may find the `getComponentCount()`, `getComponents()`, and `getComponent(int)` methods helpful.

ScrollPane

The ScrollPane container was introduced with the 1.1 release of the Java Runtime Environment (JRE) to provide a new Container with automatic scrolling of any one large Component. That large object could be anything from an image that is too big for the display area to a bunch of spreadsheet cells. All the event handling mechanisms for scrolling are managed for you. Also, there is no LayoutManager for a ScrollPane since there is only a single object within it.

The following example demonstrates the scrolling of a large image. Since an `Image` object is not a Component, the image must be drawn by a component such as a `Canvas`.



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

class ImageCanvas extends Component {
    private Image image;
    public ImageCanvas(Image i) {
        image = i;
    }
    public void paint(Graphics g) {
```

```
if (image != null)
    g.drawImage(image, 0, 0, this);
}

}

public class ScrollingImage extends Applet {
    public void init() {
        setLayout(new BorderLayout());
        ScrollPane sp =
            new ScrollPane(ScrollPane.SCROLLBARS_ALWAYS);
        Image im =
            getImage(getCodeBase(), "./images/SMarea.gif");
        sp.add(new ImageCanvas(im));
        add(sp, BorderLayout.CENTER);
    }
}
```

Event Handling

Events

Beginning with the 1.1 version of the JRE, objects register as listeners for events. If there are no listeners when an event happens, nothing happens. If there are twenty listeners registered, each is given an opportunity to process the event, in an undefined order. With a Button, for example, activating the button notifies any registered ActionListener objects. Consider SimpleButtonEvent applet which creates a Button instance and registers itself as the listener for the button's action events:

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

public class SimpleButtonEvent extends Applet
    implements ActionListener {
    private Button b;

    public void init() {
        b = new Button("Press me");
        b.addActionListener(this);
        add(b);
    }

    public void actionPerformed(ActionEvent e) {
        // If the target of the event was our Button
        // In this example, the check is not
        // truly necessary as we only listen to
        // a single button
        if ( e.getSource() == b ) {
            getGraphics().drawString("OUCH",20,20);
        }
    }
}
```

```
}
```

Notice that any class can implement ActionListener, including, in this case, the applet itself. All listeners are always notified. If you don't want an event to be processed further, you can call the AWTEvent.consume() method. However each listener would then need to check for consumption using the isConsumed() method. Consuming events primarily stops events from being processed by the system, after every listener is notified. So, if you want to reject keyboard input from the user, you can consume() the KeyEvent. All the KeyListener implementers will still be notified, but the character will not be displayed. (Consumption only works for InputEvent and its subclasses.)

So, here is how everything works:

Components generate subclasses of AWTEvent when something interesting happens.

Event sources permit any class to be a listener using the addXXXListener() method, where XXX is the event type you can listen for, for example addActionListener(). You can also remove listeners using the removeXXXListener() methods. If there is an add/removeXXXListener() pair, then the component is a source for the event when the appropriate action happens.

In order to be an event handler you have to implement the listener type, otherwise, you cannot be added, ActionListener being one such type.

Some listener types are special and require you to implement multiple methods. For instance, if you are interested in key events, and register a KeyListener, you have to implement three methods, one for key press, one for key release, and one for both, key typed. If you only care about key typed events, it doesn't make sense to have to stub out the other two methods. There are special classes out there called adapters that implement the listener interfaces and stub out all the methods. Then, you only need to subclass the adapter and override the necessary method(s).

AWTEvent:

Events subclass the AWTEvent class. And nearly every event-type has an associated Listener interface, PaintEvent and InputEvent do not. (With PaintEvent, you just override paint() and update(), for InputEvent, you listen for subclass events, since it is abstract.

Low-level Events:

Low-level events represent a low-level input or window operation, like a key press, mouse movement, or window opening. The following table displays the different low-level events, and the operations that generate each event (each operation corresponds to a method of the listener interface):

ComponentEvent	Hiding, moving, resizing, showing
ContainerEvent	Adding/removing component
FocusEvent	Getting/losing focus
KeyEvent	Pressing, releasing, or typing (both) a key
MouseEvent	Clicking, dragging, entering, exiting, moving, pressing, or releasing
WindowEvent	Iconifying, deiconifying, opening, closing, really closed, activating, deactivating

For instance, typing the letter 'A' on the keyboard generates three events, one for pressing, one for releasing, and one for typing. Depending upon your interests, you can do something for any of the three events.

Semantic Events:

Semantic events represent interaction with a GUI component; for instance selecting a button, or changing the text of a text field. Which components generate which events is shown in the next section.

ActionEvent	Do the command
AdjustmentEvent	Value adjusted
ItemEvent	State changed
TextEvent	Text changed

Event Sources:

The following table represents the different event sources. Keep in mind the object hierarchy. For instance, when Component is an event source for something, so are all its subclasses:

Low-Level Events	
Component	ComponentListener FocusListener KeyListener MouseListener MouseMotionListener
Container	ContainerListener
Window	WindowListener

Semantic Events	
Button List MenuItem TextField	ActionListener
Choice Checkbox Checkbox CheckboxMenuItem List	ItemListener
Scrollbar	AdjustmentListener
TextArea TextField	TextListener

Notice that although there is only one `MouseEvent` class, the listeners are spread across two interfaces. This is for performance issues. Since motion mouse events are generated more frequently, if you have no interest in them, you can ignore them more easily, without the performance hit.

Event Listeners:

Each listener interface is paired with one event type and contains a method for each type of event the event class embodies. For instance, the KeyListener contains three methods, one for each type of event that the KeyEvent has: keyPressed(), keyReleased(), and keyTyped().

Summary of Listener interfaces and their methods

Interface	Method(s)
ActionListener	actionPerformed(ActionEvent e)
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent e)
ComponentListener	componentHidden(ComponentEvent e)
	componentMoved(ComponentEvent e)
	componentResized(ComponentEvent e)
	componentShown(ComponentEvent e)
ContainerListener	componentAdded(ContainerEvent e)
	componentRemoved(ContainerEvent e)
FocusListener	focusGained(FocusEvent e)
	focusLost(FocusEvent e)
ItemListener	itemStateChanged(ItemEvent e)
KeyListener	keyPressed(KeyEvent e)
	keyReleased(KeyEvent e)
	keyTyped(KeyEvent e)
MouseListener	mouseClicked(MouseEvent e)
	mouseEntered(MouseEvent e)
	mouseExited(MouseEvent e)

	mousePressed(MouseEvent e)
	mouseReleased(MouseEvent e)
MouseMotionListener	mouseDragged(MouseEvent e)
	mouseMoved(MouseEvent e)
TextListener	textValueChanged(TextEvent e)
WindowListener	windowActivated(WindowEvent e)
	windowClosed(WindowEvent e)
	windowClosing(WindowEvent e)
	windowDeactivated(WindowEvent e)
	windowDeiconified(WindowEvent e)
	windowIconified(WindowEvent e)
	windowOpened(WindowEvent e)

Event Adapters:

Since the low-level event listeners have multiple methods to implement, there are event adapter classes to ease the pain. Instead of implementing the interface and stubbing out the methods you do not care about, you can subclass the appropriate adapter class and just override the one or two methods you are interested in. Since the semantic listeners only contain one method to implement, there is no need for adapter classes.

```
public class MyKeyAdapter extends KeyAdapter {
    public void keyTyped(KeyEvent e) {
        System.out.println("User typed: " +
```

```
        KeyEvent.getKeyText(e.getKeyCode()));

    }

}
```

Button Pressing Example

The following code demonstrates the basic concept a little more beyond the earlier example. There are three buttons within a Frame, their displayed labels may be internationalized so you need to preserve their purpose within a command associated with the button. Based upon which button is pressed, a different action occurs.



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

public class Activator {
    public static void main(String[] args) {
        Button b;
        ActionListener al = new MyActionListener();
        Frame f = new Frame("Hello Java");
        f.add(b = new Button("Hola"),
              BorderLayout.NORTH);
```

```
b.setActionCommand("Hello");
b.addActionListener(al);
f.add(b = new Button("Aloha"),
      BorderLayout.CENTER);

b.addActionListener(al);
f.add(b = new Button("Adios"),
      BorderLayout.SOUTH);

b.setActionCommand("Quit");
b.addActionListener(al);
f.pack();
f.show();
}

}
```

```
class MyActionListener implements ActionListener {

    public void actionPerformed(ActionEvent e) {
        // Action Command is not necessarily label
        String s = e.getActionCommand();
        if (s.equals("Quit")) {
            System.exit(0);
        }
        else if (s.equals("Hello")) {
            System.out.println("Bon Jour");
        }
    }
}
```

```
else {  
    System.out.println(s + " selected");  
}  
}  
}  
}
```

Since this is an application, you need to save the source (as Activator.java), compile it, and run it outside the browser. Also, if you wanted to avoid checking which button was selected, you can associate a different ActionListener to each button, instead of one to all. This is actually how many Integrated Development Environments (IDEs) generate their code.

Adapters Example

The following code demonstrates using an adapter as an anonymous inner class to draw a rectangle within an applet. The mouse press signifies the top left corner to draw, with the mouse release the bottom right.

```
import java.awt.*;  
  
import java.awt.event.*;  
  
public class Draw extends java.applet.Applet {  
  
    public void init() {  
        addMouseListener(  
            new MouseAdapter() {  
                int savedX, savedY;  
  
                public void mousePressed(MouseEvent e) {  
                    savedX = e.getX();  
                }  
            }  
        );  
    }  
}
```

```
    savedY = e.getY();

}

public void mouseReleased(MouseEvent e) {

    Graphics g = Draw.this.getGraphics();

    g.drawRect(savedX, savedY,
               e.getX()-savedX,
               e.getY()-savedY);

}

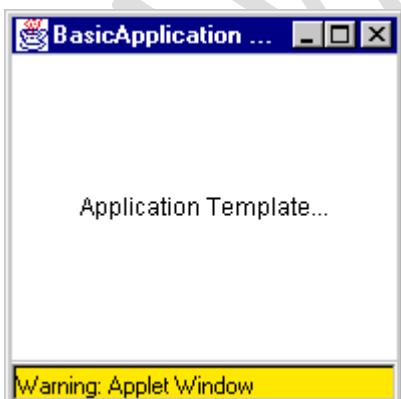
);

}

}
```

Applications and Menus GUI-based Applications

To create a window for your application, define a subclass of Frame (a Window with a title, menubar, and border) and have the main method construct an instance of that class. Applications respond to events in the same way as applets do. The following example, BasicApplication, responds to the native window toolkit quit, or closing, operation:



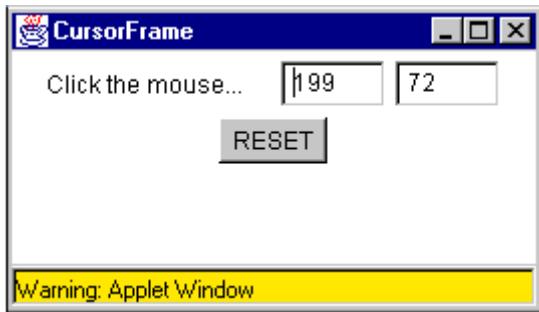
```
import java.awt.*;
import java.awt.event.*;

public class BasicApplication extends Frame {
    public BasicApplication() {
        super("BasicApplication Title");
        setSize(200, 200);
        // add a demo component to this frame
        add(new Label("Application Template...", 
            Label.CENTER),
            BorderLayout.CENTER);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                setVisible(false); dispose();
                System.exit(0);
            }
        });
    }

    public static void main(String[] args) {
        BasicApplication app =
            new BasicApplication();
        app.setVisible(true);
    }
}
```

{}

Consider an application that displays the x,y location of the last mouse click and provides a button to reset the displayed x,y coordinates to 0,0:



```
import java.awt.*;
import java.awt.event.*;
public class CursorFrame extends Frame {
    TextField a, b;
    Button btn;
    public CursorFrame() {
        super("CursorFrame");
        setSize(400, 200);
        setLayout(new FlowLayout());
        add(new Label("Click the mouse..."));
        a = new TextField("0", 4);
        b = new TextField("0", 4);
        btn = new Button("RESET");
    }
}
```

```
add(a); add(b); add(btn);

addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
        a.setText(String.valueOf(e.getX()));
        b.setText(String.valueOf(e.getY()));
    }
});

addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        setVisible(false);
        dispose();
        System.exit(0);
    }
});

btn.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        a.setText("0");
        b.setText("0");
    }
});

public static void main(String[] args) {
    CursorFrame app = new CursorFrame();
    app.setVisible(true);
}
```

```
    }  
}  
}
```

This application provides anonymous classes to handle mouse events, application window closing events, and the action event for resetting the text fields that report mouse coordinates.

When you have a very common operation, such as handling application window closing events, it often makes sense to abstract out this behavior and handle it elsewhere. In this case, it's logical to do this by extending the existing Frame class, creating the specialization AppFrame:

```
import java.awt.*;  
  
import java.awt.event.*;  
  
  
public class AppFrame extends Frame  
    implements WindowListener {  
  
    public AppFrame(String title) {  
  
        super(title);  
  
        addWindowListener(this);  
  
    }  
  
    public void windowClosing(WindowEvent e) {  
  
        setVisible(false);  
  
        dispose();  
  
        System.exit(0);  
  
    }  
  
    public void windowClosed(WindowEvent e) {}
```

```
public void windowDeactivated(WindowEvent e) {}

public void windowActivated(WindowEvent e) {}

public void windowDeiconified(WindowEvent e) {}

public void windowIconified(WindowEvent e) {}

public void windowOpened(WindowEvent e) {}

}
```

AppFrame directly implements WindowListener, providing empty methods for all but one window event, namely, the window closing operation. With this definition, applications such as CursorFrame can extend AppFrame instead of Frame and avoid having to provide the anonymous class for window closing operations:

Exercises

Displaying Files

Converting an Applet to an Application

Applications: Dialog Boxes

A Dialog is a window that requires input from the user. Components may be added to the Dialog like any other container. Like a Frame, a Dialog is initially invisible. You must call the method setVisible() to activate the dialog box.



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

```
public class DialogFrame extends AppFrame {  
  
    Dialog d;  
  
    public DialogFrame() {  
        super("DialogFrame");  
        setSize(200, 100);  
        Button btn, dbtn;  
        add(btn = new Button("Press for Dialog Box"),  
            BorderLayout.SOUTH);  
        d = new Dialog(this, "Dialog Box", false);  
        d.setSize(150, 150);  
        d.add(new Label("This is the dialog box."),  
            BorderLayout.CENTER);  
        dbtn = new Button("OK"),  
            BorderLayout.SOUTH);  
        btn.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent e) {  
                d.setVisible(true);  
            }  
        });  
        dbtn.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent e) {  
                d.setVisible(false);  
            }  
        });  
    }  
}
```

```
    }

});

d.addWindowListener(new WindowAdapter() {

    public void windowClosing(WindowEvent e) {

        d.setVisible(false);

    }

});

}

public static void main(String[] args) {

    DialogFrame app = new DialogFrame();

    app.setVisible(true);

}

}
```

Again, you can define anonymous classes on the fly for:

Activating the dialog window from the main application's command button. Deactivating the dialog window from the dialog's command button. Deactivating the dialog window in response to a native window system's closing operation.

Although the anonymous class functionality is quite elegant, it is inconvenient to have to repeatedly include the window-closing functionality for every dialog instance that your applications instantiate by coding and registering the anonymous window adapter class. As with AppFrame, you can define a specialization of Dialog that adds this functionality and thereafter simply use the enhanced class. For example, WMDialog provides this functionality:

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

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

public class WMDialog extends Dialog

    implements WindowListener {

    public WMDialog(Frame ref, String title, boolean modal) {

        super(ref, title, modal);

        addWindowListener(this);

    }

    public void windowClosing(WindowEvent e) {

        setVisible(false);

    }

    public void windowClosed(WindowEvent e) {}

    public void windowDeactivated(WindowEvent e) {}

    public void windowActivated(WindowEvent e) {}

    public void windowDeiconified(WindowEvent e) {}

    public void windowIconified(WindowEvent e) {}

    public void windowOpened(WindowEvent e) {}

}

}
```

Exercise

OK Dialog

Applications: File Dialog Boxes

The utility class `FileDialog` is useful for getting file names from the application user. `FileDialog.getFile()` reports the name of the file selected by the user. If this returns null, the user decided to cancel the selection. Currently, usage of `FileDialog` is limited to applications and trusted applets.

CALL:9010990285/7893636382

Exercise

Display a File from FileDialog

Applications: Menus

An application can have a MenuBar object containing Menu objects that are comprised of MenuItem objects. Each MenuItem can be a string, menu, checkbox, or separator (a line across the menu).

To add menus to any Frame or subclass of Frame:

Create a MenuBar

```
MenuBar mb = new MenuBar();
```

Create a Menu

```
Menu m = new Menu("File");
```

Create your MenuItem choices and add each to the Menu , in the order you want them to appear, from top to bottom.

```
m.add(new MenuItem("Open"));

m.addSeparator(); // add a separator

m.add(new CheckboxMenuItem("Allow writing"));

// Create submenu

Menu sub = new Menu("Options...");

sub.add(new MenuItem("Option 1"));

m.add(sub); // add sub to File menu
```

Add each Menu to the MenuBar in the order you want them to appear, from left to right.

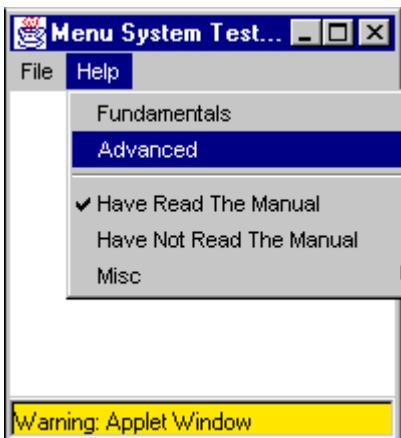
```
mb.add(m); // add File menu to bar
```

CALL:9010990285/7893636382

Add the MenuBar to the Frame by calling the setMenuBar() method.

```
setMenuBar(mb); // set menu bar of your Frame
```

The following program, MainWindow, creates an application window with a menu bar and several menus using the strategy outlined above:



```
import java.awt.*;  
  
import java.awt.event.*;  
  
// Make a main window with two top-level menus: File and Help.  
  
// Help has a submenu and demonstrates a few interesting menu items.  
  
public class MainWindow extends Frame {  
  
    public MainWindow() {  
  
        super("Menu System Test Window");  
  
        setSize(200, 200);  
  
        // make a top level File menu  
  
        FileMenu fileMenu = new FileMenu(this);
```

```
// make a top level Help menu

HelpMenu helpMenu = new HelpMenu(this);

// make a menu bar for this frame

// and add top level menus File and Menu

MenuBar mb = newMenuBar();

mb.add(fileMenu);

mb.add(helpMenu);

setMenuBar(mb);

addWindowListener(new WindowAdapter() {

    public void windowClosing(WindowEvent e) {

        exit();

    }

});

}

public void exit() {

    setVisible(false); // hide the Frame

    dispose(); // tell windowing system to free resources

    System.exit(0); // exit

}

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

```
MainWindow w = new MainWindow();

w.setVisible(true);

}

}

// Encapsulate the look and behavior of the File menu

class FileMenu extends Menu implements ActionListener {

    MainWindow mw; // who owns us?

    public FileMenu(MainWindow m) {

        super("File");

        mw = m;

        MenuItem mi;

        add(mi = new MenuItem("Open"));

        mi.addActionListener(this);

        add(mi = new MenuItem("Close"));

        mi.addActionListener(this);

        add(mi = new MenuItem("Exit"));

        mi.addActionListener(this);

    }

    // respond to the Exit menu choice

    public void actionPerformed(ActionEvent e) {

        String item = e.getActionCommand();

        if (item.equals("Exit"))

            mw.exit();
    }
}
```

```
else

    System.out.println("Selected FileMenu " + item);

}

}

// Encapsulate the look and behavior of the Help menu

class HelpMenu extends Menu implements ActionListener {

    MainWindow mw; // who owns us?

    public HelpMenu(MainWindow m) {

        super("Help");

        mw = m;

        MenuItem mi;

        add(mi = new MenuItem("Fundamentals"));

        mi.addActionListener(this);

        add(mi = new MenuItem("Advanced"));

        mi.addActionListener(this);

        addSeparator();

        add(mi = new CheckboxMenuItem("Have Read The Manual"));

        mi.addActionListener(this);

        add(mi = new CheckboxMenuItem("Have Not Read The Manual"));

        mi.addActionListener(this);

    }

    // make a Misc sub menu of Help menu

    Menu subMenu = new Menu("Misc");
```

```
subMenu.add(mi = new MenuItem("Help!!!"));

mi.addActionListener(this);

subMenu.add(mi = new MenuItem("Why did that happen?"));

mi.addActionListener(this);

add(subMenu);

}

// respond to a few menu items

public void actionPerformed(ActionEvent e) {

String item = e.getActionCommand();

if (item.equals("Fundamentals"))

    System.out.println("Fundamentals");

else if (item.equals("Help!!!"))

    System.out.println("Help!!!");

// etc...

}

}
```

Exercise

Menus

Menu Shortcuts

One nice feature of the MenuItem class is its ability to provide menu shortcuts or speed keys. For instance, in most applications that provide printing capabilities, pressing Ctrl-P initiates the printing process. When you create a MenuItem you can specify the shortcut associated with it. If the user happens to press the speed key, the action event is triggered for the menu item.

The following code creates two menu items with speed keys, Ctrl-P for Print and Shift-Ctrl-P for Print Preview:

```
file.add (mi = new MenuItem ("Print",
    new MenuShortcut('p')));

file.add (mi = new MenuItem ("Print Preview",
    new MenuShortcut('p', true)));
```

The example above uses Ctrl-P and Shift-Ctrl-P shortcuts on Windows/Motif. The use of Ctrl for the shortcut key is defined by the Toolkit method getMenuShortcutKeyMask(). For the Macintosh, this would be the Command key. An optional boolean parameter to the constructor determines the need for the Shift key appropriate to the platform.

Pop-up Menus

One restriction of the Menu class is that it can only be added to a Frame. If you want a menu in an Applet, you are out of luck (unless you use the Swing component set). While not necessarily a perfect solution, you can associate a pop-up menu with any Component, of which Applet is a subclass. A PopupMenu is similar to a Menu in that it holds MenuItem objects. However, instead of appearing at the top of a Frame, you pop the popup menu up over any component, usually when the user generates the appropriate mouse event.

The actual mouse interaction to generate the event is platform specific so there is the means to determine if a MouseEvent triggers the pop-up menu using the MouseEvent.isPopupTrigger() method. It is then your responsibility to position and display the PopupMenu.

The following program, PopupApplication, demonstrates this portable triggering of a pop-up menu, as well as activating a pop-up menu from a command button:



```
import java.awt.*;
import java.awt.event.*;
public class PopupApplication extends AppFrame {
    Button btn; TextField msg; PopupAppMenu m;
    public PopupApplication() {
        super("PopupApplication");
        setSize(200, 200);
        btn = new Button("Press for pop-up menu...");
        add(btn, BorderLayout.NORTH);
        msg = new TextField();
        msg.setEditable(false);
        add(msg, BorderLayout.SOUTH);
        m = new PopupAppMenu(this);
        add(m);
        btn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                m.show(btn, 10, 10);
            }
        });
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                if (e.isPopupTrigger())
                    m.show(e.getComponent(), e.getX(), e.getY());
            }
        });
    }
}
```

```
}

public void mouseReleased(MouseEvent e) {
    if (e.isPopupTrigger())
        m.show(e.getComponent(), e.getX(), e.getY());
}
});

}

public static void main(String[] args) {
    PopupApplication app = new PopupApplication();
    app.setVisible(true);
}
}

class PopupAppMenu extends PopupMenu
    implements ActionListener {
    PopupApplication ref;
    public PopupAppMenu(PopupApplication ref) {
        super("File");
        this.ref = ref;
        MenuItem mi;
        add(mi = new MenuItem("Copy"));
        mi.addActionListener(this);
        add(mi = new MenuItem("Cut"));
        mi.addActionListener(this);
    }
}
```

```
add(mi = new MenuItem("Paste"));

mi.addActionListener(this);

}

public void actionPerformed(ActionEvent e) {

String item = e.getActionCommand();

ref.msg.setText("

    Selected menu item: " + item);

}

}
```

SWINGS

- In the earlier days SUN micro system; we have a concept called awt.
- awt is used for creating GUI components.
- All awt components are written in 'C' language and those components appearance is changing from one operating system to another operating system. Since, 'C' language is the platform dependent language.
- In later stages SUN micro system has developed a concept called swings.
- Swings are used for developing GUI components and all swing components are developed in java language.

- Swing components never change their appearance from one operating system to another operating system. Since, they have developed in platform independent language.

Differences between awt and swings:

Awt

1. awt components are developed in 'C' language.
2. All awt components are platform dependent.
3. All awt components are heavy weight components, since whose processing time and main memory space is more.

Swings

1. Swing components are developed in java language.
2. All swing components are platform independent.
3. All swing components are light weight components, since its processing time and main memory space is less.

NOTE: All swing components in java are preceded with a letter 'J'.

For example:

```
JButton JB=new JButton ("ok");
```

All components of swings are treated as classes and they are belongs to a package called

`javax.swing.*`

In swings, we have two types of components. They are auxiliary components and logical components.

- Auxiliary components are those which we can touch and feel. For example, mouse, keyboard, etc.
- Logical components are those which we can feel only. Logical components are divided into two types. They are passive or inactive components and active or interactive components.
- Passive components are those where there is no interaction from user. For example, JLabel.
- Active components are those where there is user interaction. For example, JButton, JCheckbox, JRadioButton, etc.

- > In order to provide functionality or behavior to swing GUI active components one must import a package called `java.awt.event.*`
- > This package contains various classes and interfaces which provides functionality to active components.
- > EDM is one which always provides the functionality to GUI active components.

Steps in EDM:

1. Every GUI active component can be processed in two ways. They are based on name or label of the component and based on reference of the component. Whenever we interact with any GUI component whose reference and label will be stored in one of the predefined class object whose general notation is `xxxEvent` class.

For example:

`JButton _ ActionEvent`

`JCheckbox _ ItemEvent`

2. In order to provide behavior of the GUI component we must write some statements in methods only. And these methods are given by SUN micro system without definition. Such type of methods is known as abstract methods. In general, all abstract methods present in interfaces and those interfaces in swings known as Listeners. Hence, each and every interactive component must have the appropriate Listener whose general notation is `xxxListener`.

For example:

`JButton -> ActionListener`

`JCheckbox -> ItemListener`

3. Identify the abstract methods which are present in `xxxListener` to provide functionality to GUI component by overriding the abstract method.

For example:

`JButton ActionListener -> public abstract void actionPerformed (ActionEvent)`

`JCheckbox ItemListener -> public abstract void itemStateChanged (ItemEvent)`

4. Every GUI interactive component must be registered with appropriate Listener. Each interactive component will have the following generalized method to register or unregister with appropriate Listener.

```
public void addXxxListener (XxxListener) _ Registration
```

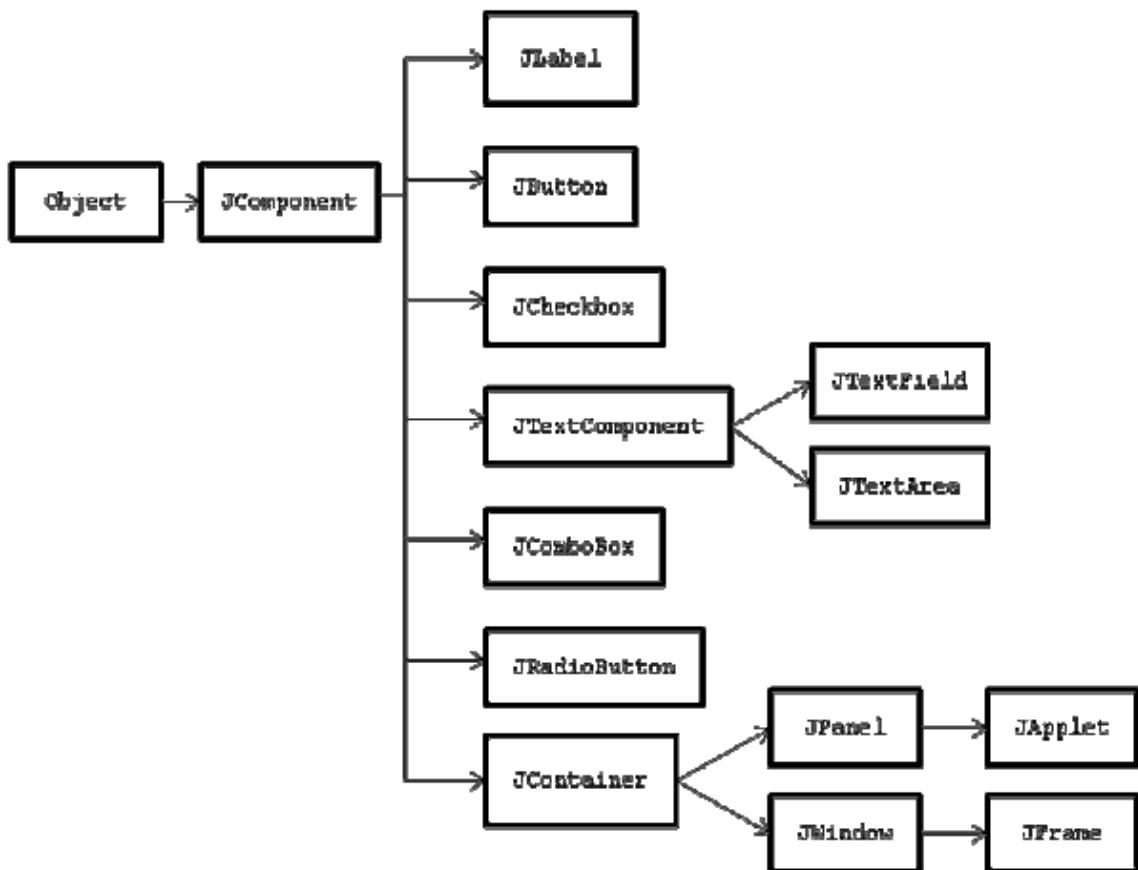
```
public void removeXxxListener (XxxListener) _ Unregistration
```

For example:

```
public void addActionListener (ActionListener) _ Registration
```

```
public void removeActionListener (ActionListener) _ Unregistration
```

Hierarchy chart in Swings:



NOTE: Creating any component is nothing but creating an object of appropriate swing component class.

Q) Develop the following application:

The application window has a title bar labeled "Calculator". Inside, there are four text input fields and four buttons. The first two fields are labeled "Enter first number:" and "Enter second number:", each containing the value "10" and "20" respectively. The third field is labeled "Result:" and is currently empty. Below these fields are three buttons labeled "Sum", "Sub", and "Mul". At the bottom left is another button labeled "Exit".

Answer:

CALL:9010990285/7893636382

```
import javax.swing.*;  
import java.awt.*;  
import java.awt.event.*;  
  
class Op extends Frame implements ActionListener  
{  
    JLabel jl1, jl2, jl3;  
    JTextField jtf1, jtf2, jtf3;  
    JButton jb1, jb2, jb3, jb4;  
  
    Op ()  
    {  
        setTitle ("Operations");  
        setSize (200, 200);  
        setLayout (new FlowLayout ());  
        jl1=new JLabel ("Enter first number: ");  
        jl2=new JLabel ("Enter second number: ");  
        jl3=new JLabel ("Result: ");  
        jtf1=new JTextField (20);  
        jtf2=new JTextField (20);  
        jtf3=new JTextField (20);  
        jb1=new JButton ("Sum");  
        jb2=new JButton ("Sub");  
        jb3=new JButton ("Mul");  
        jb4=new JButton ("Exit");  
  
        add (jl1);add (jtf1);  
        add (jl2);add (jtf2);
```

```
add (jl3);add (jtf3);

add (jb1);add (jb2);add (jb3);add (jb4);

jb1.addActionListener (this);

jb2.addActionListener (this);

jb3.addActionListener (this);

jb4.addActionListener (this);

setVisible (true);

}

public void actionPerformed (ActionEvent ae)

{

if (ae.getSource ()==jb1)

{

String s1=jtf1.getText ();

String s2=jtf2.getText ();

int n3=Integer.parseInt (s1)+Integer.parseInt (s2);

String s3=String.valueOf (n3);

jtf3.setText (s3);

}

if (ae.getSource ()==jb2)

{

String s1=jtf1.getText ();

String s2=jtf2.getText ();

int n3=Integer.parseInt (s1)-Integer.parseInt (s2);

String s3=String.valueOf (n3);

jtf3.setText (s3);

}
```

```
}

if (ae.getSource ()==jb3)

{

String s1=jtf1.getText ();

String s2=jtf2.getText ();

int n3=Integer.parseInt (s1)*Integer.parseInt (s2);

String s3=String.valueOf (n3);

jtf3.setText (s3);

}

if (ae.getSource ()==jb4)

{

System.exit (0);

}

}

}

class OpDemo

{

public static void main (String [] args)

{

Op o1=new Op ();

}}
```

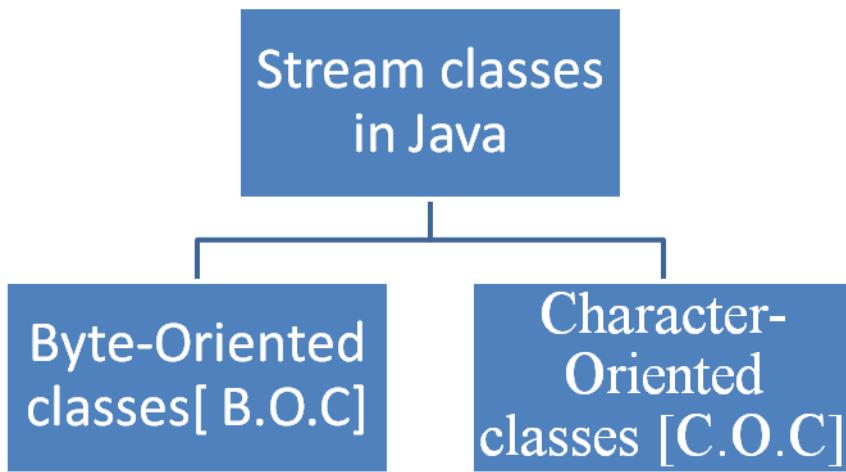
CHAPTER16

STREAMS

Streams in Java

->A stream is a flow of data between two devices. Any data passed between two devices of a computer will be in the form of streams. To represent streams, java language supports a package called “java.io”.

-> The classes defined in this package can be broadly classified as follows:



-> Any Stream class that ends with the word "Stream" is a Byte-Oriented class and any class that ends either with the words "Reader" or "Writer" is a Character-Oriented class.

-> The purpose of streams is to transfer data from one place to another. InputStream classes mention the address of the Input device (i.e., source) and OutputStream classes mention the address of the Output device (i.e., destination).

--->For Example, in the statement, System.out.println(), "out" is the reference of the PrintStream class which contains the address of destination output device i.e., monitor. Using println() method, we can transfer data anywhere, i.e., console devices or files or programs on other systems.

->But, we cannot directly create the object of PrintStream class because the constructor of it needs to access arguments of output stream. To receive or send data, we need the address of source or destination. Addresses are represented in the form of objects. In order to represent address of destination, we use output stream class. Using Input stream class, we represent the address of source. Consider the class System as follows :

```
class System
{
    public static PrintStream out = new PrintStream(new OutputStream(){anonymous class});
}
```

Now, we can call System.out.println("Hello"); where "out" contains OutputStream object which contains the address of VDU or console.

Reading data from console (or anywhere):

-> Though there are many methods available for inputting data, it is better to use String class objects to read input. For Example, `readLine()` is a method that reads data from resource whose address is specified. It is a non-static method defined in both Byte and Character-Oriented classes and return-type is String. Using this method, we can read both character and byte oriented data.

->For Example consider the following program:

// This program is used to read a string

```
import java.io.*;

class Reading

{
    public static void main(String args[])
    {
        /* pass address of keyboard to the constructor of DataInputStream class as an object of
        InputStream class.In the System class, "in" is a static reference of InputStream class that
        contains the address of console */

        DataInputStream dis = new DataInputStream(System.in);

        System.out.println("Enter Your Name ");

        String s = dis.readLine(); // readLine() returns input as a String

        System.out.println("Welcome " + s);

    }// end of main

}// end of class
```

->But, it is noted that the `readLine()` method in the above program is defined to throw Exception, so it should be defined in try-catch block. But, when we use try-catch block, the compiler will display a message that we are using a deprecated method. So, it will be better using `readLine()` method in `java.io.BufferedReader` class.

NOTE: The constructor of `BufferedReader` class does not accept object of `InputStream` class. Instead, it accepts the object of `InputStreamReader` class. The constructor of the `InputStreamReader` class is defined to accept the object of `InputStream` class.

->The following program demonstrates the use of readLine() method of BufferedReader class.

```
/* Program to accept input from keyboard through readLine() of BufferedReader.  
*/  
  
import java.io.*;  
  
class ReadInput  
{  
  
    public static void main(String args[]) throws Exception  
{  
  
        InputStreamReader isr = new InputStreamReader(System.in);  
  
        BufferedReader br = new BufferedReader(isr);  
  
        System.out.println("Enter some text ");  
  
        String msg = br.readLine();  
  
        System.out.println("The entered text is " + msg);  
  
    }  
}
```

Eg : 2)

```
/* Program to accept two values from keyboard and display their sum */
```

```
import java.io.*;  
  
class InputNumber  
{  
  
    public static void main(String args[]) throws Exception  
{  
  
        InputStreamReader isr = new InputStreamReader(System.in);  
  
        BufferedReader br = new BufferedReader(isr);  
  
        System.out.println("Enter First Number ");  
  
        CALL:9010990285/7893636382
```

```
int a = Integer.parseInt(br.readLine());  
System.out.println("Enter Second Number ");  
int b = Integer.parseInt(br.readLine());  
int c = a + b;  
System.out.println("Sum is " + c);  
}  
}
```

File Handling in Java:

->A file is the name of memory location where we can store data permanently under computer's hard disk. Java language supports a set of classes in java.io package to create and maintain files.

->The following are some of the classes we can use to create and maintain files :

--- FileInputStream

--- FileOutputStream

--- FileReader
--- FileWriter
--- RandomAccessFile etc.,

FileOutputStream class : This class represent data as a stream of bytes. The objects of this class can be used to open and access a file in write mode. The object for this class can be created as below :

```
FileOutputStream fis = new FileOutputStream("file-name",boolean-value);
```

->Here, if the boolean value is true, the file can be opened in append mode. If it is false, the data is overwritten from the beginning of the file every time when we execute the program. The following program demonstrates the use of this class.

```
/* Program to store data into a file using FileOutputStream class */
```

```
import java.io.*;  
  
class FileWrite  
{  
  
    public static void main(String args[]) throws Exception  
    {  
  
        InputStreamReader isr = new InputStreamReader(System.in);  
        BufferedReader br = new BufferedReader(isr);  
  
        System.out.println("Enter some text ");  
  
        String data = br.readLine();  
  
        FileOutputStream fos = new FileOutputStream("file1.txt",true);  
  
        /* Because data is in the form of String object, convert it into byte array */  
  
        byte b[] = data.getBytes();  
  
        fos.write(b); /* writes byte array into file.txt */  
    }  
}
```

```
fos.close();  
}  
}
```

FileInputStream class: This class represents data as a stream of bytes. The objects of this class can be used to open and access a file in read mode. The object for this class can be created as below :

```
FileInputStream fis = new FileInputStream("file-name");
```

-> The following program demonstrates the use of this class.

```
/* Program to read data from a file using FileInputStream class */  
  
import java.io.*;  
  
class FileRead  
{  
  
    public static void main(String args[]) throws Exception  
{  
  
        FileInputStream fis = new FileInputStream("file1.txt");  
  
        /* Read the size of the file */  
  
        int size = fis.available(); /* available() returns no. of bytes in file1.txt */  
  
        byte b [] = new byte[size]; /* create a byte array of file size */  
  
        fis.read(b);  
  
        String data = new String(b); /* convert the byte array to string */  
  
        System.out.println("The contents of file1.txt are : " + data);  
  
    }// end of main  
  
}// end of class
```

Eg : 2)

CALL:9010990285/7893636382

```
/* The following program is used to read data from any file */

import java.io.*;

class FileRead2

{

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

    {

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        String ch="yes";

        while(ch.equals("yes"))

        {

            System.out.println("Enter the file name to Read ");

            String fname = br.readLine();

            FileInputStream fis = new FileInputStream(fname);

            int size = fis.available(); /* available() returns no. of bytes in file1.txt */

            byte b [] = new byte[size]; /* create a byte array of file size */

            fis.read(b);

            String data = new String(b); /* convert the byte array to string */

            System.out.println("The contents of file1.txt are : " + data);

            fis.close();

            System.out.println("Want to Continue(type yes/no) ");

            ch = br.readLine();

        } // end while

    } // end main

} // end class
```

FileWriter class: This class represent data as a stream of characters. The objects of this class can be used to open and access a file in write mode. The object for this class can be created as below :

```
FileWriter fw = new FileWriter(filename,boolean-value);
```

->**The following program demonstrates the use of FileWriter class.**

```
import java.io.*;  
  
class FileWrite  
{  
  
    public static void main(String args[]) throws Exception  
{  
  
        FileWriter fw = new FileWriter("file2.txt",true);  
  
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));  
  
        System.out.println("Enter some text ");  
  
        String data = br.readLine();  
  
        // convert data into equalent character array  
  
        char ch[] = data.toCharArray();  
  
        fw.write(ch);  
  
        fw.close();  
    } // end main  
}  
} // end class
```

FileReader class: This class represent data as a stream of characters. The objects of this class can be used to open and access a file in read mode. The object for this class can be created as below :

```
FileReader fr = new FileReader(filename);
```

->**The following program demonstrates the use of FileReader class.**

```
import java.io.*;  
  
class FileRead  
{  
  
    public static void main(String args[]) throws Exception  
{  
  
        FileReader fr = new FileReader("file2.txt");  
  
        String data = " ";  
  
        int i = fr.read(); // read first character  
  
        while(i != -1) // not end-of-file  
  
        {  
  
            char ch = (char)i; //convert into character  
  
            data = data + ch; // append to string data  
  
            i = fr.read();  
  
        }  
  
        System.out.println("The contents of file2.txt are : " + data);  
  
        fr.close();  
    } // end of main  
  
} // end of class
```

NOTE: The problem of using the above stream classes to create files is that FileOutputStream class stores data as an array of bytes while FileWriter class stores data as an array of characters.

-> But, in most real-time applications files are used to store data as a set of records. That is, assume an Employee data file. This file needs to store emp_id as integer, name as string, basic as float etc., But, the above classes are not used to represent data in the form of their corresponding data types. To overcome this problem, we can use RandomAccessFile which represents data as its equivalent Primitive data types.

->The following program demonstrates the use of RandomAccessFile class

```
/* Program to implement Random Access Files */

import java.io.*;

class RandomReadWrite

{

int empno;

String name;

float basic;

RandomAccessFile raf;

BufferedReader br;

RandomReadWrite() // constructor

{

try{

raf = new RandomAccessFile("emp.dat","rw"); // rw—read & write mode

br = new BufferedReader(new InputStreamReader(System.in));

}

catch(Exception e){System.out.println(e);}

public void readEmpData()

{

try{

System.out.println("Enter Employee Number ");

empno = Integer.parseInt(br.readLine());

System.out.println("Enter Employee Name ");

name = br.readLine();

System.out.println("Enter Employee Salary ");

basic = Float.parseFloat(br.readLine());

}
```

```
        }

    catch(Exception e){System.out.println(e);}

} // end of readEmpData()

public void writeEmpFile()

{

    int size = raf.length(); // find the size of file

    raf.seek(size); // move cursor to end of file

    String ch = "yes";

    while(ch.equals("yes"))

    {

        readEmpData();

        raf.writeInt(empno) ;

        raf.writeUTF(name);

        raf.writeFloat(basic);

        System.out.println("Any more records(yes/no) ");

        ch = br.readLine();

    } // end while

} // end of WriteEmpFile()

public void readEmpFile()

{

    raf.seek(0); // place cursor at the start of file

    int size = raf.length();

    while(raf.getFilePointer()<size)

    {

        int id = raf.readInt();
```

```
String na = raf.readUTF();
float sal = raf.readFloat();
System.out.println(id + " " + na + " " + sal);
} // end while
raf.close();
} // end of ReadEmpFile()

public static void main(String args[])
{
    RandomReadWrite r = new RandomReadWrite();
    r.writeEmpFile();
    r.readEmpFile();
} // end of main
} // end of class
```

Need for RandomAccess Files:

-> Now-a-days, every business data is represented in the form of databases. So, there is no need for creating files using programming languages like C,C++,or Java. But, RandomAccessFile concept can be used when we need to migrate data available in a file which is developed some years back, into a table(or database).

GENERICs

JDK 1.5 provides one of the several extentions to the java programming language i.e. the "generics". This allows you to abstract over types. Here, in this program the code given in the program

IdentityHashMap<Integer, String> map = **new** IdentityHashMap<Integer, String>();
determines the new version of the program fragment of old typed program using "generics".

In this code, IdentityHashMap is a generic class that takes type parameters for identifying the retrieved value whether it is false or true. This program uses the put() method of the **IdentityHashMap** class which puts the two types values one is the integer value and another is the string type value. So, this program firs store the passed type parameter with the **IdentityHashMap** class and its constructor i.e. the <Integer, String>.

Here is the code of program:

```
import java.io.*;  
import java.util.*;  
  
public class AssociateValue{
```

```
public static void main(String[] args) throws IOException{
    try{
        String str;
        IdentityHashMap<Integer,String> map = new IdentityHashMap<Integer,String>();
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter the hash code for the first object: ");
        int a = Integer.parseInt(in.readLine());
        System.out.print("Enter the text value for this code: ");
        map.put(new Integer(a), in.readLine());
        System.out.print("Enter the hash code for the second object: ");
        int b = Integer.parseInt(in.readLine());
        System.out.print("Enter the text value for this code: ");
        map.put(new Integer(b), in.readLine());
        System.out.print("Enter the hash code for the third object: ");
        int c = Integer.parseInt(in.readLine());
        System.out.print("Enter the text value for this code: ");
        map.put(new Integer(c), in.readLine());
        System.out.print("Enter the hash code for the fourth object: ");
        int d = Integer.parseInt(in.readLine());
        System.out.print("Enter the text value for this code: ");
        map.put(new Integer(d), in.readLine());
        System.out.println(map);
        Set set = map.entrySet();
        Iterator it = set.iterator();
        while(it.hasNext()){
            Map.Entry me = (Map.Entry)it.next();
            System.out.print(me.getKey() + ": ");
            System.out.println(me.getValue());
        }
    }
    catch(NumberFormatException ne){
        System.out.println(ne.getMessage() + " is not a legal value.");
        System.exit(0);
    }
}
```

Comparing Arrays : Java Util

CALL:9010990285/7893636382

This section show you how to determine the given arrays are same or not. The given program illustrates you how to compare arrays according to the content of the that.

In this section, you can see that the given program initializes two arrays and input five number from user through the keyboard. And then the program checks whether the given taken both arrays are same or not. This comparison operation is performed by using the equals() method of **Arrays** class.

Arrays.equals():

Above method compares two arrays.

Arrays is the class of the `java.util.*;` package. This class and it's methods are used for manipulating arrays.

Here is the code of the program:

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

public class ComparingArrays{
    public static void main(String[] args) throws IOException{
        int[] array1 = new int[5];
        int[] array2 = new int[5];
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        try{
            System.out.println("Enter 5 numbers for the first Array : ");
            for(int i = 0; i < array1.length; i++){
                array1[i] = Integer.parseInt(in.readLine());
            }
            System.out.println("Enter 5 numbers for the second Array : ");
            for(int i = 0; i < array2.length; i++){
                array2[i] = Integer.parseInt(in.readLine());
            }
        } catch(NumberFormatException ne){
            ne.printStackTrace();
        }

        boolean check = Arrays.equals(array1, array2);
```

```
if(check == false)
    System.out.println("Arrays are not same.");
else
    System.out.println("Both Arrays are same.");
}
}
```

Shuffling the Element of a List or Array

Shuffling is the technique i.e. used to randomize the list or array to prepare each and every element for the operation. In this section, you will learn about shuffling the element of a list or array. Shuffling the element means moving the element from one place to another place randomly.

In the given program, list is created which contains some values. List lists the elements of the Collection. This program takes some values for the list and it sorts these values after shuffling. For this purposes some methods and APIs are explained below which have been used in the program:

List:

This is the class of `java.util.*;` package which extends Collection. This is used to list elements available in the Collections. In this program a argument `<Integer>` has been passed with the List for type checking.

ArrayList():

This is the construction of **ArrayList** class of the `java.util.*;` package that extends the **AbstractList** class. It constructs an empty list. You can specify the capacity of the list by passing a value to the constructor.

sort(list):

This is the method of the **Collections** class which is used to sort the list elements in the default order.

Here is the code of program:

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

public class ShufflingListAndArray{
```

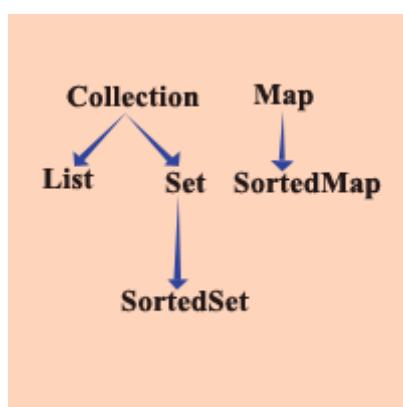
```
public static void main(String[] args) throws IOException{
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("How many elements you want to add to the list: ");
    int n = Integer.parseInt(in.readLine());
    System.out.print("Enter " + n + " numbers to sort: ");
    List<Integer> list = new ArrayList<Integer>();
    for(int i = 0; i < n; i++){
        list.add(Integer.parseInt(in.readLine()));
    }
    Collections.shuffle(list);
    Collections.sort(list);
    System.out.println("List sorting :" + list);
}
```

COLLECTIONS FRAMEWORK

Java provides the **Collections Framework**. In the Collection Framework, a collection represents the group of the objects. And a collection framework is the unified architecture that represent and manipulate collections. The collection framework provides a standard common programming interface to many of the most common abstraction without burdening the programmer with many procedures and interfaces. It provides a system for organizing and handling collections. This framework is based on:

- Interfaces that categorize common collection types.
- Classes which proves implementations of the Interfaces.
- Algorithms which proves data and behaviors need when using collections i.e. search, sort, iterate etc.

Following is the hierarchical representation for the relationship of all four interfaces of the collection framework which illustrates the whole implementation of the framework.



During the designing of a software (application) it need to be remember the following relationship of the four basic interfaces of the framework are as follows:

- The Collection interface which is the collection of objects. That permits the duplication of the value or objects.
- Set is the interface of the collections framework which extends the Collection but forbids duplicates.
- An another interface is the List which also extends the Collection. It allows the duplicates objects on different position because it introduces the positional indexing.
- And Map is also a interface of the collection framework which extends neither Set nor Collection.

Some interfaces and classes of the collection framework are as follows:

INTERFACES	CLASSES	
Collection	InterfaceHashSet	Class
Iterator	InterfaceTreeSet	Class
Set	InterfaceArrayList	Class
List	InterfaceLinkedList	Class
ListIterator	InterfaceHashMap	Class
Map	InterfaceTreeMap	Class
SortedSet	InterfaceVector	Class
SortedMap Interface	Stack Class	

Advantage of the Collections Framework:

The collections framework offers developers the following benefits:

- It increases the readability of your collections by providing a standard set of interfaces which has to be used by many programmers in different applications.
- It makes your code more flexible. You can make the code flexible by using many interfaces and classes of the collection framework.
- It offers many specific implementations of the interfaces. It allows you to choose the collection that is most fitting and which offers the highest performance for your purposes.

Interfaces of the collections framework are very easy to use. These interfaces can be transparently substituted to increase the developed application performance.

Collection Iterate Example

In this section, you will get the detailed explanation about the **hasNext()** method of interface **Iterator**. We are going to use **hasNext()** method of interface **Iterator** in Java. The description of the code is given below for the usage of the method.

Description of the code:

Here, you will get to know about the **hasNext()** method through the following java program. True is return by this method in case the iteration has more elements. This means that if the iteration has more elements then the **hasNext()** method will return true rather than throwing an exception.

In the program code given below, we have taken a string of elements. We have converted this string of elements into a list of array and then we have applied the **hasNext()** method which returns true because there are more elements in the list. Hence we get the following output.

Here is the code of program:

```
import java.util.*;  
  
public class hasNext{  
    public static void main (String args[]){  
        boolean b;  
        String elements[] = {"Blue", "Grey", "Teal"};  
        Set s = new HashSet(Arrays.asList(elements));  
        Iterator i = s.iterator();  
        if (b = i.hasNext()) {  
            System.out.println(b);  
        }  
    }  
}
```

Java Next()

CALL:9010990285/7893636382

In this section, you will get the detailed explanation about the **next()** method of interface **Iterator**. We are going to use **next()** method of interface **Iterator** in Java. The description of the code is given below for the usage of the method.

Description of the code:

Here, you will get to know about the **next()** method through the following java program. This method returns next element in the iteration in case the iteration has more than one element. This means that if the iteration has more than one element then the **next()** method will return the next element. However, if no more element exist then it throws **NoSuchElementException**.

In the program code given below, we have taken a string of elements. We have converted this string of elements into a list of array and then we have applied the **hasNext()** method first which will check if there is another element or not and if there exist another element then it will return that (next element) with the help of **next()** method. Hence we get the following output.

Here is the code of program:

```
import java.util.*;  
  
public class next{  
    public static void main (String args[]) {  
        String elements[] = {"Blue", "Grey", "Teal"};  
        Set s = new HashSet(Arrays.asList(elements));  
        Iterator i = s.iterator();  
        while (i.hasNext()) {  
            System.out.println(i.next());  
        }  
    }  
}
```

Java remove()

In this section, you will get the detailed explanation about the **remove()** method of interface **Iterator**. We are going to use **remove()** method of interface **Iterator** in Java. The description of the code is given below for the usage of the method.

Description of the code:

Here, you will get to know about the **remove()** method through the following java program. This method removes the last element returned by the iterator (optional operation) from the collection. As per every call to next, this method can only be called once. However, the iteration will continue other than by calling this method. This method will throw

UnsupportedOperationException if the iterator doesn't support the remove operation and will throw **IllegalStateException** if after the last call to the next method, the remove method has already been called.

In the program code given below, we have taken a list of array that contains 0- 10 elements. Then with the help of **hasNext()** method we will get next elements as output. And to remove any element from the loop, we have applied **remove()** method which will remove that element which gives 0 remainder in this iteration. Hence we get the following output.

Here is the code of program:

```
import java.util.*;  
  
public class remove{  
    public static void main (String args[]){  
        ArrayList arr = new ArrayList();  
        System.out.println("Total elements of iterator: ");  
        for (int i = 1; i < 10; i++) arr.add(i);  
        int count = 0;  
        for (Iterator i = arr.iterator(); i.hasNext();){  
            System.out.println ("element is " + i.next());  
        }  
        System.out.println("After removing : ");  
        for (Iterator i = arr.iterator(); i.hasNext();){  
            count++;  
            i.next();  
            if (count%4 == 0) i.remove();  
        }  
        for (Iterator i = arr.iterator(); i.hasNext();){  
            System.out.println ("element is " + i.next());  
        }  
    }  
}
```

```
}
```

Java nextElement()

In this section, you will get the detailed explanation about the **nextElement()** method of interface **Enumeration**. We are going to use **nextElement()** method of interface **Enumeration** in Java. The description of the code is given below for the usage of the method.

Description of the code:

Here, you will get to know about the **nextElement()** method through the following java program. This method returns next element of the enumeration in case the enumeration has more than one element. This means that if the enumeration has more than one element then the **nextElement()** method will return the next element. However, if no more element exist then it throws **NoSuchElementException**.

In the program code given below, we have taken a Vector of string type. Then we have applied the **hasMoreElement()** method which will check for the next element and if it will find the next element then it will return that with the help of **nextElement()** method.

Here is the code of program:

```
import java.util.Enumeration;
import java.util.Vector;
import java.util.*;

public class nextElement{
    public static void main (String[] args){

        Vector strVector = new Vector();
        String str = new String();
        strVector.addElement(new String("Hi"));
        strVector.addElement(new String("Hello"));
        strVector.addElement(new String("Namaste"));
        strVector.addElement(new String("Salam"));

    }
}
```

```
Enumeration elements = strVector.elements();
while (elements.hasMoreElements()){
    str = (String)elements.nextElement();
    System.out.println(str);
}
```

Converting Collection to an Array

Here is the illustration for the conversion from the collection to an array. In this section, you will learn how to do this. The given example gives you a brief introduction for convert collection to an array without losing any data or element present in the collection.

This program creates a List. List is a type of collection that contains a ordered list of elements. This list (collection) is converted into an array. Length of the created array is defined by the system according to the number of elements in the list. Each and every subscripts hold separate element.

Code Description:

Here is the explanation of some methods and APIs whatever has been used in the program:

List:

List is the class of the `java.util` package which creates a list for containing some elements. In this program, this class is used with the type checking.

add():

Above method adds elements to the list. This method holds string which is the element name.

List.toArray(new String[0]):

Above code converts a list into a String Array. This method returns a string array which length is fixed according to the number of elements present in the list. Overall this code converts collection to an array.

Here is the code of the program:

```
import java.util.*;  
  
public class CollectionToArray{  
    public static void main(String[] args){  
        List<String> list = new ArrayList<String>();  
        list.add("This ");  
        list.add("is ");  
        list.add("a ");  
        list.add("good ");  
        list.add("boy.");  
        String[] s1 = list.toArray(new String[0]); //Collection to array  
        for(int i = 0; i < s1.length; ++i){  
            String contents = s1[i];  
            System.out.print(contents);  
        }  
    }  
}
```

Converting an Array to a Collection

Here is the illustration for the conversion from the an array to a collection. In this section, you will learn how how to do this. The given example gives you a brief introduction for converting an array to collection without losing any data or element held by the array.

This program creates an array, that contains all the elements entered by you. This array is converted into a list (collection). Every value for the separate subscript of the array will be converted as elements of the list (collection).

Code Description:

Arrays.asList(name):

Above code converts an array to a collection. This method takes the name of the array as a parameter which has to be converted.

for(String

li:

list):

This is the for-each loop. There are two arguments mentioned in the loop. In which, first is the String type variable li another one is the list name which has to be converted. This loop check each and every elements of the list and store it into the mentioned variable li.

Here is the code of the program:

```

import java.util.*;
import java.io.*;

public class ArrayToCollection{
    public static void main(String args[]) throws IOException{
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("How many elements you want to add to the array: ");
        int n = Integer.parseInt(in.readLine());
        String[] name = new String[n];
        for(int i = 0; i < n; i++){
            name[i] = in.readLine();
        }
        List<String> list = Arrays.asList(name); //Array to Collection
        for(String li: list){
            String str = li;
            System.out.print(str + " ");
        }
    }
}

```

Collection to Array:

The example given below illustrates the conversion of a collection into a array. In this example we creating an object of **ArrayList**, adding elements into this object and storing this object into **List** interface. Convert elements of **ArrayList** into an array by using **toArray()**.

List interface is a member of the Java **Collection** Framework and extends **Collection** interface. It is an ordered collection which follows insertion order, typically allow duplicate elements. The class **ArrayList** extends **AbstractList** class and implements **List**, **RandomAccess**, **Cloneable**, **Serializable** interfaces. **ArrayList** class is a member of the Java Collections Framework.

The **toArray()** method used:

toArray(Object[] a): This method returns an array containing all of the elements of the given list.

The code of the program is given below:

CALL:9010990285/7893636382

```
import java.util.List;
import java.util.ArrayList;

public class CollectionToArray1{
    public static void main(String[] args){
        List arraylist = new ArrayList();
        arraylist.add("Java");
        arraylist.add("Struts");
        arraylist.add("J2ee");
        arraylist.add("Code");
        Object[] array = arraylist.toArray();
        System.out.println("\nThe objects into array.");
        for (int i = 0; i < array.length; i++)
        {
            System.out.println(array[i].toString());
        }
    }
}
```

Implement the Queue in Java:

In this section, you will learn how to implement the queue. A queue holds a collection of data or elements and follows the **FIFO** (First In First Out) rule. The FIFO that means which data added first in the list, only that element can be removed or retrieved first from the list. In other sense, You can remove or perform operation on that data which had been added first in the Collection (list). Whenever you need to remove the last added element then you must remove all these elements which are entered before the certain element.

The given program implements a queue. It takes all elements as input by user. These values are added to the list and shows first, last and the rest elements present in the list separately. Some methods and APIs are explained below which have been used in the program for the certain purposes:

LinkedList<Integer>():

This is the constructor of the **LinkedList** class. This class is used by importing the `java.util.*;` package. This constructor is used for constructing an empty list. It can contain integer types data in the given program because in the declaration of the **LinkedList** class type checking has been used. This is an implementation of the **List** interface of the

Collections Framework. The **LinkedList** class provides inserting and deleting the data to/from the list.

removeFirst():

Above method removes and returns the first element of the list.

removeLast():

Above method removes and returns the last element of the list.

list.isEmpty():

This method checks whether the list is empty or not.

remove():

This method used to remove the elements in the list in a specified sequence.

Here is the code of program:

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

public class QueueImplement{
    LinkedList<Integer> list;
    String str;
    int num;

    public static void main(String[] args){
        QueueImplement q = new QueueImplement();
    }

    public QueueImplement(){
        try{
            list = new LinkedList<Integer>();
            InputStreamReader ir = new InputStreamReader(System.in);
            BufferedReader bf = new BufferedReader(ir);
            System.out.println("Enter number of elements : ");
            str = bf.readLine();
            if((num = Integer.parseInt(str)) == 0){
                System.out.println("You have entered either zero/null.");
                System.exit(0);
            }
            else{

```

```
System.out.println("Enter elements : ");
for(int i = 0; i < num; i++){
    str = bf.readLine();
    int n = Integer.parseInt(str);
    list.add(n);
}
System.out.println("First element : " + list.removeFirst());
System.out.println("Last element : " + list.removeLast());
System.out.println("Rest elements in the list :");
while(!list.isEmpty()){
    System.out.print(list.remove() + "\t");
}
}
catch(IOException e){
    System.out.println(e.getMessage() + " is not a legal entry.");
    System.exit(0);
}
}
```

Implementing a Stack in Java:

In this section, you will learn how to implement a stack in Java. A Stack is like a bucket in which you can put elements one-by-one in sequence and retrieve elements from the bucket according to the sequence of the last entered element. Stack is a collection of data and follows the **LIFO** (Last in, first out) rule that mean you can insert the elements one-by-one in sequence and the last inserted element can be retrieved at once from the bucket. Elements are inserted and retrieved to/from the stack through the push() and pop() method.

This program implements a stack and follows the LIFO rule. It asks you for the number of elements which have to be entered in the stack and then it takes elements for insertion in the stack. All the elements present in the stack are shown from the last inserted element to the first inserted element.

Stack<Integer>():

Above constructor of the **Stack** class creates a empty stack which holds the integer type

value which is mention with the creation of stack. The **Stack** class extends the **Vector** class and both classes are implemented from the `java.util.*;` package.

Stack.push(Object

obj:

Above method is used to insert or push the data or element in the stack. It takes an object like: data or elements and push its onto the top of the stack.

Stack.pop():

This is the method to removes the objects like: data or elements at the top positions of stack.

Here is the code of program:

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

public class StackImplement{
    Stack<Integer> stack;
    String str;
    int num, n;
    public static void main(String[] args){
        StackImplement q = new StackImplement();
    }
    public StackImplement(){
        try{
            stack = new Stack<Integer>();
            InputStreamReader ir = new InputStreamReader(System.in);
            BufferedReader bf = new BufferedReader(ir);
            System.out.print("Enter number of elements : ");
            str = bf.readLine();
            num = Integer.parseInt(str);
            for(int i = 1; i <= num; i++){
                System.out.print("Enter elements : ");
                str = bf.readLine();
                n = Integer.parseInt(str);
                stack.push(n);
            }
        }
    }
}
```

```
catch(IOException e){}
System.out.print("Retrieved elements from the stack : ");
while (!stack.empty()){
    System.out.print(stack.pop() + " ");
}
}
```

JAVA BY SATEESH

REFLECTION API

"Reflection is the process of obtaining runtime information about the class or interface."

CALL:9010990285/7893636382

Runtime information is nothing but deal with the following:

1. Finding the name of the class or interface.
2. Finding the data members of the class or interface.
3. Finding number of constructors (default constructor and number of parameterized constructors).
4. Number of instance methods.
5. Number of static methods.
6. Determining modifiers of the class (modifiers of the class can be public, final, public + final, abstract and public + abstract).
7. Obtaining super class of a derived class.
8. Obtaining the interfaces which are implemented by various classes.

Real time applications of reflection:

1. Development of language complier, debuggers, editors and browsers.
2. In order to deal with reflection in java, we must import a predefined package called `java.lang.reflect.*`
3. The package reflect contains set of predefined classes and interfaces which are used by the programmer to develop reflection applications. Number of ways to obtain runtime information about a class (or) number of ways to get an object of a class called `Class`:

The predefined class called `Class` is present in a package called `java.lang.Class` (fully qualified name of a class called `Class` is `java.lang.Class`). In java we have 4 ways to deal with or to create an object of `java.lang.Class`, they are:

- 1) When we know that class name at compile time and to get runtime information about the class, we must use the following:

```
Ex1: Class c=c1.class (c1 - user defined class)  
Ex2: Class c=java.lang.String.class (String - predefined class)
```

- 2) When we know the object name at runtime, to get the class name or class type of the runtime object, we must use the following:

```

Ex1: c1 o1=new c1 ();
        Class c=o1.getClass ();
Ex2: String s=new String ("HELLO");
        Class c=s.getClass ();
    
```

getClass is the predefined method present in a predefined class called java.lang.Object and whose prototype is given below:

```

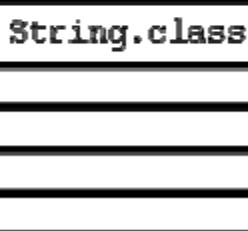
java.lang.Object
↓
public Class getClass () → instance method
Here, public - access specifier & Class - return type
    
```

3) When an object is given at runtime, we must find runtime information about current class and its super class.

```

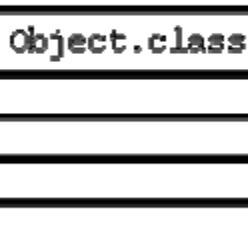
String s=new String ("HELLO");
Class c=s.getClass ();
    
```

C



```

Class sc=c.getSuperclass ();
    sc
    
```



Q) Write a java program to print name of the current class and its super class name?

Answer:

CALL:9010990285/7893636382

```
class First
{
    public static void main (String [] args)
    {
        String s=new String ("HELLO");
        printSuperclass (s);
    }
    static void printSuperclass (Object s)
    {
        Class c=s.getClass ();
        Class sc=c.getSuperclass ();
        System.out.println ("NAME OF CURRENT CLASS : "+c.getName ());
        System.out.println ("NAME OF THE SUPER CLASS : "+sc.getName ());
    }
};
```

Output:

```
java First
NAME OF CURRENT CLASS : java.lang.String
NAME OF THE SUPER CLASS : java.lang.Object
```

4) We know the class name at runtime and we have to obtain the runtime information about the class.

To perform the above we must use the method `java.lang.Class` and whose prototype is given below:

```
java.lang.Class -->
public static Class forName (String) throws ClassNotFoundException
```

When we use the forName as a part of java program it performs the following operations:

- It can create an object of the class which we pass at runtime.
- It returns runtime information about the object which is created.

For example:

```
try
{
    Class c=Class.forName ("java.awt.Button");
}
catch (ClassNotFoundException cnfe)
{
    System.out.println ("CLASS DOES NOT EXIST...");
}
```

forName is taking String as an argument. If the class is not found forName method throws an exception called ClassNotFoundException. Here, forName method is a factory method (a factory method is one which return type is similar to name of the class where it presents).

Every factory method must be static and public. The class which contains a factory method is known as Singleton class (a java class is said to be Singleton class through which we can create single object per JVM).

For example:

java.lang.Class is called Singleton class

Write a java program to find name of the class and its super class name by passing the class name at runtime?

Answer:

```
class ref1
{
    public static void main (String [] args)
    {
        if (args.length==0)
        {
            System.out.println ("PLEASE PASS THE CLASS NAME..!");
        }
        else
        {
            try
            {
                Class c=Class.forName (args [0]);
                printSuperclass (c);
            }
            catch (ClassNotFoundException cnfe)
            {
                System.out.println (args [0]+" DOES NOT EXISTS...");
            }
        }// else
    }// main

    static void printSuperclass (Class c)
    {
        String s=c.getName ();
        Class sc=c.getSuperclass ();
    }
}
```

```
String sn=sc.getName();  
  
System.out.println (sn+" IS THE SUPER CLASS OF "+s);  
  
}// printSuperclass  
  
}// ref1
```

Output:

```
java ref1 java.awt.TextField  
  
java.awt.TextComponent IS THE SUPER CLASS OF java.awt.TextField
```

Q) Write a java program to print super class hierarchy at a current class which is passed from command prompt?

Answer:

```
class Hierarchy  
{  
public static void main (String [] args)  
{  
if (args.length==0)  
{  
System.out.println ("PLEASE PASS THE CLASS NAME..!");  
}  
else  
{  
try
```

```
{  
Class c=Class.forName (args [0]);  
printHierarchy (c);  
}  
catch (ClassNotFoundException cnfe)  
{  
System.out.println (args [0]+" DOES NOT EXISTS...");  
}  
}  
}  
}  
}  
}  
static void printHierarchy (Class c)  
{  
Class c1=c;  
String cname=c1.getName ();  
System.out.println (cname);  
Class sc=c1.getSuperclass ();  
while (sc!=null)  
{  
cname=sc.getName ();  
System.out.println (cname);  
c1=sc;  
sc=c1.getSuperclass ();  
}  
}  
};  
};
```

Output:

```
java Hierarchy java.awt.TextField
java.awt.TextField
java.awt.TextComponent
java.awt.Component
java.lang.Object
```

Obtaining information about CONSTRUCTORS which are present in a class:

In order to get the constructor of the current class we must use the following method:

```
java.lang.Class ——————→
public Constructor [] getConstructors ()
```

For example:

```
Constructor cons []=c.getConstructors ();
System.out.println ("NUMBER OF CONSTRUCTORS = "+cons.length);
```

In order to get the parameters of the constructor we must use the following method:

```
java.lang.reflect.Constructor ——————→
public Class [] getParameterTypes ()
```

For example:

```
Class ptype []=cons [0].getParameterTypes ();
```

Write a java program to obtain constructors of a class?

Answer:

```
class ConsInfo
```

```
{
```

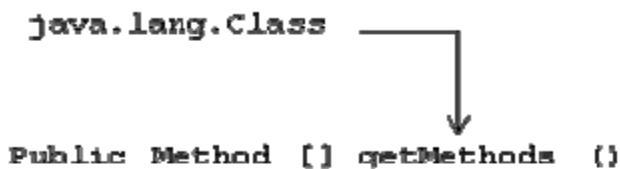
```
CALL:9010990285/7893636382
```

```
public static void main (String [] args)
{
if (args.length==0)
{
System.out.println ("PLEASE PASS THE CLASS NAME..!");
}
else
{
try
{
Class c=Class.forName (args [0]);
printConsts (c);
}
catch (ClassNotFoundException cnfe)
{
System.out.println (args [0]+" DOES NOT EXISTS...");
}
}
static void printConsts (Class c)
{
java.lang.reflect.Constructor Cons []=c.getConstructors ();
System.out.println ("NUMBER OF CONSTRUCTORS = "+Cons.length);
System.out.println ("NAME OF THE CONSTRUCTOR : "+c.getName());
for (int i=0; i<Cons.length; i++)
}
```

```
{  
System.out.print (c.getName ()+"(");  
Class cp []=Cons [i].getParameterTypes ();  
for (int j=0; j<cp.length; j++)  
{  
System.out.print (cp [j].getName ()+")");  
}  
System.out.println ("\b"+")");  
}  
}  
};
```

Obtaining METHODS information:

In order to obtain information about methods we must use the following methods:



For example:

```
Method m []=c.getMethods ();  
System.out.println ("NUMBER OF METHODS = "+m.length);
```

Associated with methods we have return type of the method, name of the method and types of parameters passed to a method.

The Method class contains the following methods:

1. public Class getReturnType ();
2. public String getName ();

3. public Class [] getParameterTypes ();

Method-1 gives return type of the method, Method-2 gives name of the method and

Method-3 gives what parameters the method is taking.

Q) Write a java program to obtain information about methods which are present in a class?

Answer:

```
import java.lang.reflect.*;

class MetInfo

{

public static void main (String [] args)

{

try

{

if (args.length==0)

{



System.out.println ("PLEASE PASS THE CLASS NAME..!");

}

else

{



Class c=Class.forName (args [0]);

printMethods (c);

}

}

catch (ClassNotFoundException cnfe)

{

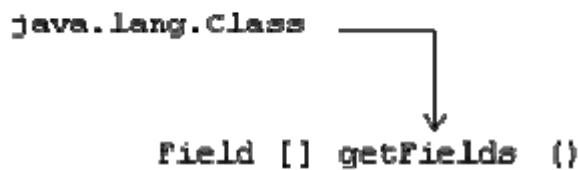


}
```

```
System.out.println (args [0]+" DOES NOT EXISTS...");  
}  
}  
  
static void printMethods (Class c)  
{  
Method m []=c.getMethods ();  
System.out.println ("NUMBER OF METHODS = "+m.length);  
System.out.println ("NAME OF THE CLASS : "+c.getName ());  
for (int i=0; i<m.length; i++)  
{  
Class c1=m [i].getReturnType ();  
String rtype=c1.getName ();  
String mname=m [i].getName ();  
System.out.print (rtype+" "+mname+"(");  
Class mp []=m [i].getParameterTypes ();  
for (int j=0; j<mp.length; j++)  
{  
String ptype=mp [j].getName ();  
System.out.print (ptype+",");  
}  
System.out.println ("\b"+")");  
}  
}  
};
```

Obtaining FIELDS or DATA MEMBERS of a class:

In order to obtain information about fields or data members of the class we must use the following method.



For example:

```
Field f []=c.getFields ();
System.out.println ("NUMBER OF FIELDS = "+f.length);
```

Associated with field or data member there is a data type and field name:

The Field class contains the following methods:

1. public Class getType ();
2. public String getName ();

Method-1 is used for obtaining data type of the field and Method-2 is used for obtaining name of the field.

Write a java program to print fields or data members of a class?

Answer:

```
import java.lang.reflect.Field;
class Fields
{
    void printFields (Class c)
    {
        Field f []=c.getFields ();
        System.out.println ("NUMBER OF FIELDS : "+f.length);
        for (int i=0; i<f.length; i++)
    }
```

```
{  
String fname=f [i].getName ();  
Class s=f [i].getType ();  
String ftype=s.getName ();  
System.out.println (ftype+" "+fname);  
}  
}  
};  
  
class FieldsDemo  
{  
public static void main (String [] args)  
{  
if (args.length==0)  
{  
System.out.println ("PLEASE PASS THE CLASS NAME..!");  
}  
else  
{  
try  
{  
Class c=Class.forName (args [0]);  
Fields fs=new Fields ();  
fs.printFields (c);  
}  
catch (ClassNotFoundException cnfe)  
{  
}
```

```
{  
System.out.println (args [0]+"NOT FOUND...");  
}  
}  
}  
};
```

JAVA BY SATEESH