



SRI VENKATESHWARAA COLLEGE OF ENGINEERING & TECHNOLOGY

(Approved by AICTE, New Delhi & Affiliated to Pondicherry University, Puducherry.)
13-A, Villupuram – Pondy Main road, Ariyur, Puducherry – 605 102.
Phone: 0413-2644426, Fax: 2644424 / Website: www.svcetpondy.com

Department of Computer Science and Engineering

Subject Name: **OBJECT ORIENTED PROGRAMMING**

Subject Code: **CS T45**

UNIT 1 2 MARKS

1. LIST OUT THE LIMITATIONS OF C++

- It is not pure object oriented language.
- Does not provide very strong type-checking.
- C++ code is easily prone to errors related to data types, their conversions.
- Does not provide efficient means for garbage collection, as already mentioned.
- No built in support for threads

2. DEFINE JAVA AND ITS FEATURES

- Java is a pure object oriented general purpose language.
- A general-purpose computer programming language designed to produce programs that will run on any computer system.
- Java is a programming language and computing platform first released by Sun Microsystems in 1995.
- There are lots of applications and websites that will not work unless you have Java installed, and more are created every day. Java is fast, secure, and reliable

3. WHAT ARE THE FEATURES OF JAVA LANGUAGE?

- The features of java are:
 - Simple, small and familiar.
 - Object Oriented.
 - Distributed
 - Robust
 - Secure
 - Architectural neutral (or) platform independent

Portable
 Compiled and interpreted
 High performance
 Dynamic and extensible

4. HISTORY OF JAVA?

James Gosling initiated the Java language project in June 1991 for use in one of his many set-top box projects. The language, initially called Oak after an oak tree that stood outside Gosling's office, also went by the name Green and ended up later being renamed as Java, from a list of random words.

Sun released the first public implementation as Java 1.0 in 1995. It promised Write Once, Run Anywhere (WORA), providing no-cost run-times on popular platforms.

On 13 November 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL).

On 8 May 2007, Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.

5. DEFINE JDK AND JRE

JAVA RUNTIME ENVIRONMENT

Provides the backbone for running Java application.

Is a collection of software.

Allows a computer system to run a Java application.

Consists of

- JVMs, Java Virtual Machines, interpret Java *bytecode* into machine code.

- Standard class libraries

- User interface toolkits

- A variety of utilities.

JAVA DEVELOPMENT KIT

Provides all of the components and necessary resources to develop Java applications.

Is a programming environment for compiling, debugging, and running Java applets, applications, and Java Beans.

Includes the JRE, Java Programming language, development tools and tool APIs.

6. DEFINE JVM

JAVA VIRTUAL MACHINE:

An abstract computing machine, or virtual machine, JVM is a platform-independent execution environment that converts Java bytecode into machine language and executes it. Most programming languages compile source code directly into machine code that is designed to run on a specific microprocessor architecture or operating system, such as Windows or UNIX

7. GIVE THE COMMAND USED FOR CREATING, COMPILING AND EXECUTION OF A PROGRAM

Java is a command driven language. So the programs are compiled and executed by giving commands in command prompt.

Creating a program:

Create a program using any text editor such as edit in DOS or notepad or word pad etc. and save it in java directory.

Syntax:

filename.java

Compiling the program:

Compile the created program using java compiler.

Syntax:

javac sourcefilename.java

Running the program:

Run the compiled program using java interpreter.

Syntax:

java classname

EXAMPLE:

Name of the source file- Sample.java

Name of the class file- Sample.class

To run the program-c>java sample ←

8. WHAT IS BYTECODE?

Bytecode is a highly optimized set of instructions designed to be executed by the java run-time system. Which is called the java virtual machine (JVM)? JVM is an interpreter for bytecode.

9. What is a Literal? What are the different types of literals?

A Literal represents a value of a certain type where the type describes the behaviors of the value. The different types of literals are:

Number literals

Character literals

Boolean literals

String literals

10. What is a Character literals?

Character literals are expressed by a single character enclosed within single quotes. Characters are stored as Unicode characters.

Escape	Meaning
\n	Newline
\t	Tab
\b	Backspace
\r	Carriage
\f	return
\\	Form feed

11. What is a String literals?

A string is a combination of characters. string literals are a set of characters that are enclosed within double quotes. As they are real objects, it is possible to concatenate, modify and test them. For example, "This is a test string" represents a string. Strings can contain character constants and Unicode characters

12. WHAT ARE JAVA TOKENS?

A token is an individual element in java. A program is written by using the available tokens. The various tokens are:

- Keywords
- Identifies
- Constants or literals
- Operators
- Separators

13. WHAT ARE KEYWORDS (OR) DEFINE KEYWORD WITH EG:

Keywords are words which belong to java language. They have standard predefined meaning. The users have no right to change its meaning. Keywords should be written in lowercase.

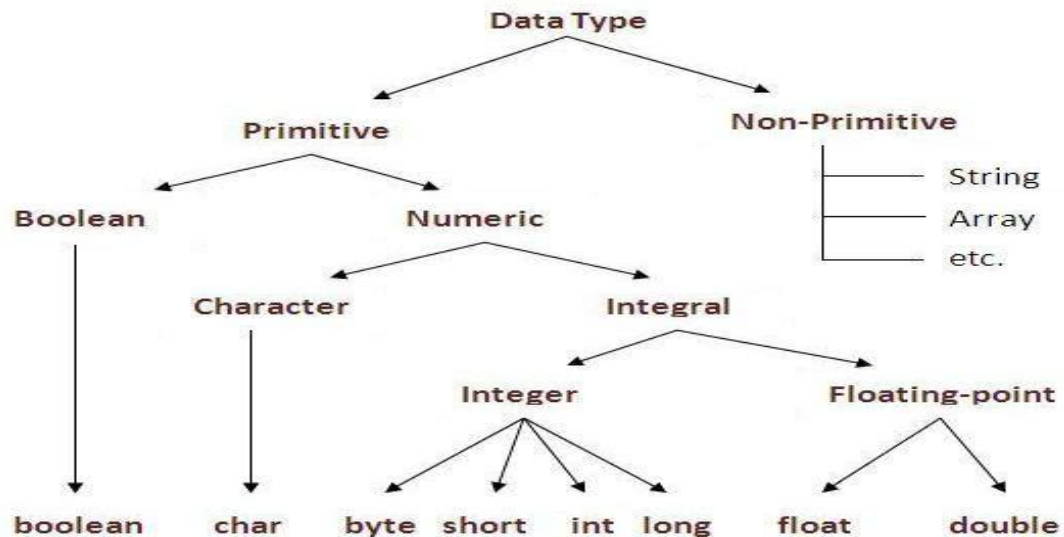
The list of keywords are: abstract, boolean, int, float, static, byte, char, for, if,... etc.

14. WHAT ARE THE DATA TYPES IN JAVA

Data types specify the size and type of value that are to be stored. The data types are defined into two categories.

They are:

- Primitive (or) built-in data types
- Derived (or) reference data types



15. DEFINE TYPE CASTING AND ITS TYPES?

Type casting in Java is to cast one type, a class or interface, into another type i.e. another class or interface.

Casting is used when a method returns a type different than the one we require.

SYNTAX:

```
type variable1 = (type) variable2;
```

Ex:

```
int m = 50;
byte n = (byte)m;
long count = (long) m;
```

TYPES:

Widening
Narrowing

16. HOW DO YOU KNOW IF AN EXPLICIT OBJECT CASTING IS NEEDED?

If you assign a superclass object to a variable of a subclass's data type, you need to do explicit casting.

For example:

```
Object a; Customer b; b = (Customer) a;
```

When you assign a subclass to a variable having a superclass type, the casting is performed automatically

17. DIFFERENTIABLE BETWEEN BREAK AND CONTINUE STATEMENTS?

The break keyword halts the execution of the current loop and forces control out of the loop. The term break refers to the act of breaking out of a block of code. Continue is similar to break, except that instead of halting the execution of the loop, it starts the next iteration.

18. DEFINE OPERATORS AND ITS TYPES IN JAVA

An operator is a symbol that operates on one or more arguments to produce a result.
Java provides a rich set of operators to manipulate variables

TYPES OF OPERATORS

- Assignment Operators
- Increment Decrement Operators
- Arithmetic Operators
- Bitwise Operators
- Relational Operators
- Logical Operators
- Ternary Operators
- Comma Operators
- Instanceof Operators

19. DEFINE OBJECT

An object is an real world entity that has its own properties (or) state and actions (or) behavior

EX:

a pen, a person, a student

20. DEFINE CLASS

A class is a template/blue print of objects those have similar properties (or) state and actions (or) behavior

Ex:

Pen, Person, Student

21. STATE THE CLASS MODEL

```
<access-specifiers> class class_name
{
    access-specifiers data-type variable-name1;
    access-specifiers data-type variable-name2;
    access-specifiers return-type method_name1(arg1,arg2,arg3)
    {
        .....
        .....
    }
    access-specifiers return-type method_name2(arg1)
    {
        .....
        .....
    }
}
```

22. DEFINE OBJECT CREATION IN JAVA

Syntax:

```
class_name object_name=new class_name();
```

It has two parts:

object declaration

Associate an object name with its class type to reserve proper amount of memory for that object

```
class_name object_name; --> Student s1;
```

object instantiation

Creation of object using new operator in the heap memory and return the reference to the stack memory

```
object_name=new class_name(); --> s1=new Student();
```

23. WHAT IS OBJECT REFERENCE?

When we declare a variable, we can use a class name as a type. Such variable refers to as the objects of that class

These variables are known as object references

24. DEFINE THE DOT(.) OPERATOR

The dot (.) operator is used to access the instance variables and methods within an object.

General Form:

Object reference (.) Variable name

Object Reference is a reference to an object and variable name is the name of the instance.

Ex:

```
p.x = 10;
```

```
p.y=20;
```

25. DEFINE VARIABLES AND ITS TYPES IN JAVA

Variable is name of reserved area allocated in memory. Ex: int data=50; //data is variable
Types of variable:

Local Variable:

A variable that is declared inside the method is called local variable

Instance Variable:

A variable that is declared inside the class but outside the method is called instance variable.
It is not declared as static

Static Variable:

A variable that is declared as static is called static variable. It is common to all objects

Example:

```
class A
{
    int data=50;//instance variable
    static int m=100;//static variable
    void method()
    {
        int n=90; //local variable
    }
} //end of class
```

26. DEFINE METHOD AND ITS SYNTAX IN JAVA

A Java method is a collection of statements that are grouped together to perform an operation
Methods are also known as Procedures or Functions

Syntax:

Method definition consists of a method header and a method body

```
access-specifier returnType nameOfMethod (Parameter List)
{
    // method body
}
```

Example:

```
public void add(int a, int b)
{
    System.out.println(a+b);}
}
```

27. STATE THE CHARACTERISTICS OF CONSTRUCTORS IN JAVA

Constructor in java is a *special type of method* that is used to initialize the object.

Java constructor is *invoked at the time of object creation*

Constructor name must be same as its class name

Constructor must have no explicit return type

Constructor can be overloaded

28. RULES FOR CREATING JAVA CONSTRUCTOR

There are basically two rules defined for the constructor.

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

29. WHAT IS THE PURPOSE OF DEFAULT CONSTRUCTOR?

Default constructor provides the default values to the object like 0, null etc. depending on the type

30. DEFINE PACKAGES IN JAVA?

Packages provide a mechanism for grouping a variety of classes and / or interfaces together.

Grouping is based on functionality

Types of packages:

Pre-defined packages or Java API packages

User-defined packages

31. DEFINE THE SYNTAX OF DEFINING PACKAGE IN JAVA

Syntax:

```
package [PackageName];
public class [ClassName]
{
    //Body of the class
}
```

Example:

```
package firstPackage;
public class FirstClass
{
    //Body of the class
}
```

32. DEFINE ACCESS SPECIFIERS AND ITS TYPES IN JAVA?

Access specifiers specifies the type of access levels for the classes, variables, methods and constructor

Types of access-specifiers:

Default - Visible to the package, the default.

Private - Visible to the class only.

Public - Visible to the world.

Protected -Visible to the package and all subclasses.

33. DEFINE METHOD OVERLOADING IN JAVA?

If a class have multiple methods by same name but different parameters, it is known as **Method Overloading**

Method Overloading increases the readability of the program

There are two ways to overload the method in java

By changing number of arguments

By changing the data type

34. STATE DIFFERENCE BETWEEN CONSTRUCTOR AND METHODS

Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.

Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

35.WHAT IS CONSTRUCTOR OVERLOADING?

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists

The compiler differentiates these constructors in two ways
the different number of parameters
the different type of parameters

36.DEFINE NEW OPERATOR

The 'new' operator in java is responsible for the creation of new object and dynamically allocates memory in the heap with the reference pointed from the stack memory.

37.DEFINE DESTRUCTOR (OR) FINALIZE METHOD

The finalize method is called when an object is about to get garbage collected. That can be at any time after it has become eligible for garbage collection

38.WHAT IS THE FINALIZE METHOD DO?

Before the invalid objects get garbage collected, the JVM give the user a chance to clean up some resources before it got garbage collected

39.WHAT IS THE FINAL KEYWORD DENOTES?

Final keyword denotes that it is the final implementation for that method or variable or class. You can't override that method/variable/class any more

40.DEFINE INTERFACE

An **interface in java** is a blueprint of a class. It has static constants and abstract methods only. It is used to achieve fully abstraction and multiple inheritance in Java. It cannot be instantiated just like abstract class

41.SYNTAX OF DEFINING INTERFACE

```
interface interfaceName
{
    Variables declaration;
    Method declaration;
}
```

42.WHAT ARE THE USES OF INTERFACE

It is used to achieve fully abstraction.

By interface, we can support the functionality of multiple inheritance.

It can be used to achieve loose coupling

43.DIFFERENCE BETWEEN CLASS AND INTERFACE

CLASS	INTERFACE
The members of a class can be constant or variables.	The members of an interface are always declared as constant , i.e., their values are final.
The class definition can contain the code for each of its methods. That is , the methods can be abstract or non-abstract.	The methods in an interface are abstract in nature , i.e., there is no code associated with them. It is later defined by the class that implements the interface.
It can be instantiated by declaring objects.	It cannot be used to declare objects. It can only be inherited by a class.
It can use various access specifiers like public , private or protected.	It can only use the public access specifier.

44.WHAT'S THE DIFFERENCE BETWEEN AN INTERFACE AND AN ABSTRACT CLASS?

An abstract class may contain code in method bodies, which is not allowed in an interface. With abstract classes, you have to inherit your class from it and Java does not allow multiple inheritance. On the other hand, you can implement multiple interfaces in your class

45.DEFINE INTERNALIZATION

Internationalization is a mechanism to create such an application that can be adapted to different languages and regions.

Internationalization is one of the powerful concepts of java if you are developing an application and want to display messages, currencies, date, time etc. according to the specific region or language

46.WHAT IS LOCALE?

A Locale object represents a specific geographical, political, or cultural region

47.STATE THE TYPES OF CONSTRUTOR

There are two types of constructors:

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

48.EXPLAIN THE USAGE OF JAVA PACKAGES

This is a way to organize files when a project consists of multiple modules. It also helps resolve naming conflicts when different packages have classes with the same names. Packages access level also allows you to protect data from being used by the non-authorized classes.

49.CAN YOU CALL ONE CONSTRUCTOR FROM ANOTHER IF A CLASS HAS MULTIPLE CONSTRUCTORS

Yes. Use this() syntax

50. HOW CAN A SUBCLASS CALL A METHOD OR A CONSTRUCTOR DEFINED IN A SUPERCLASS?

Use the following syntax: `super.myMethod()`; To call a constructor of the super class, just write `super()`; in the first line of the subclass's constructor



SRI VENKATESHWARAA COLLEGE OF ENGINEERING & TECHNOLOGY

(Approved by AICTE, New Delhi & Affiliated to Pondicherry University, Puducherry.)
13-A, Villupuram – Pondy Main road, Ariyur, Puducherry – 605 102.
Phone: 0413-2644426, Fax: 2644424 / Website: www.svcetpondy.com

Department of Computer Science and Engineering

Subject Name: **OBJECT ORIENTED PROGRAMMING**

Subject Code: **CS T45**

UNIT 1 11 MARKS

1. WRITE ABOUT JAVA AND ITS FEATURES

Java is an object-oriented, multi-threaded programming language developed by Sun Microsystems in 1991. It is designed to be small, simple and portable across different platforms as well as operating systems.

The popularity of java is due to its unique technology that is designed on the basis of three key elements. They are the usage of applets, powerful programming language constructs and rich set of significant object classes.

When a program is compiled, it is translated into machine code or processor instructions that are specific to the processor. In the Java development environment there are two parts: a Java compiler and a Java Interpreter. The compiler generates bytecode (asset of instructions that resemble machine code but are not specific to any processor) instead of machine code and the interpreter executes the Java program.

The disadvantage of using bytecode is the execution speed. Since system specific programs run directly on the hardware, they are faster than the Java bytecodes that is processed by the interpreter. In order to write a Java program, an editor, a Java compiler and a Java Runtime Environment are needed.

Java is guaranteed to be Write Once, Run Anywhere.

History of Java:

James Gosling initiated the Java language project in June 1991 for use in one of his many set-top box projects. The language, initially called Oak after an oak tree that stood outside Gosling's

office, also went by the name Green and ended up later being renamed as Java, from a list of random words.

Sun released the first public implementation as Java 1.0 in 1995. It promised Write Once, Run Anywhere (WORA), providing no-cost run-times on popular platforms.

On 13 November 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL).

On 8 May 2007, Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.

As of December 2008, the latest release of the Java Standard Edition is 6 (J2SE). With the advancement of Java and its widespread popularity, multiple configurations were built to suite various types of platforms. Ex: J2EE for Enterprise Applications, J2ME for Mobile Applications.

Sun Microsystems has renamed the new J2 versions as Java SE, Java EE and Java ME respectively.

Features of Java:

Object Oriented: In Java, everything is an Object. Java can be easily extended since it is based on the Object model.

Platform independent: Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.

Simple: Java is designed to be easy to learn. If you understand the basic concept of OOP Java would be easy to master.

Secure: With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.

Architectural-neutral : Java compiler generates an architecture-neutral object file format which makes the compiled code to be executable on many processors, with the presence of Java runtime system.

Portable: Being architectural-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary which is a POSIX subset.

Robust: Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.

Multithreaded: With Java's multithreaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.

Interpreted: Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light weight process.

High Performance: With the use of Just-In-Time compilers, Java enables high performance.

Distributed: Java is designed for the distributed environment of the internet.

Dynamic: Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

NEWLY ADDED FEATURES IN JAVA 2:

SWING is a set of user interface components that is implemented entirely in java. The user can use a look and feel that is either specific to a particular operating systems.

Collections are group of objects. Java provides several types of collection, such as linked lists, dynamic arrays, and hash tables, for our use. Collections offer a new way to solve several common programming problems.

Various tools such as javac, java and javadoc have been enhanced. Debugger and profiler interfaces for the JVM are available.

Performance improvements have been made in several areas. A Just-In-Time (JIT) compiler is included in the JDK.

Policy files can define the permission for code from various sources. These determine if a particular file or directory may be accessed, or if a connection can be established to a specific host and port.

Digital certificates provide a mechanism to establish the identity of a user, which can be referred as electronic passports.

The Java 2D library provides advanced features for working with shapes, images and text.

Various security tools are available that enable the user to create and store cryptographic keys and digital certificates, sign Java Archive (JAR) files, and check the signature of a JAR file.

The user can now play audio files such as MIDI, AU, WAV and RMF files using Java programs

2. EXPLAIN IN DETAIL ABOUT JAVA PLATFORM

The computer world currently has many platforms, among them Microsoft Windows, Macintosh, OS/2, UNIX® and NetWare®; software must be compiled separately to run on each

platform. The binary file for an application that runs on one platform cannot run on another platform, because the binary file is machine specific.

The Java Platform is a software platform for delivering and running highly interactive, dynamic, and secure applets and applications on networked computer systems. But what sets the Java Platform apart is that it sits on top of these other platforms, and compiles to *bytecodes*, which are not specific to any physical machine, but are machine instructions for a *virtual machine*.

A program written in the Java Language compiles to a bytecode file that can run wherever the Java Platform is present, on *any* underlying operating system. In other words, the same exact file can run on any operating system that is running the Java Platform. This portability is possible because at the core of the Java Platform is the Java Virtual Machine.

The Java Platform is therefore ideal for the Internet, where one program should be capable of running on any computer in the world.

The Java Platform is designed to provide this “Write Once, Run Anywhere” capability.

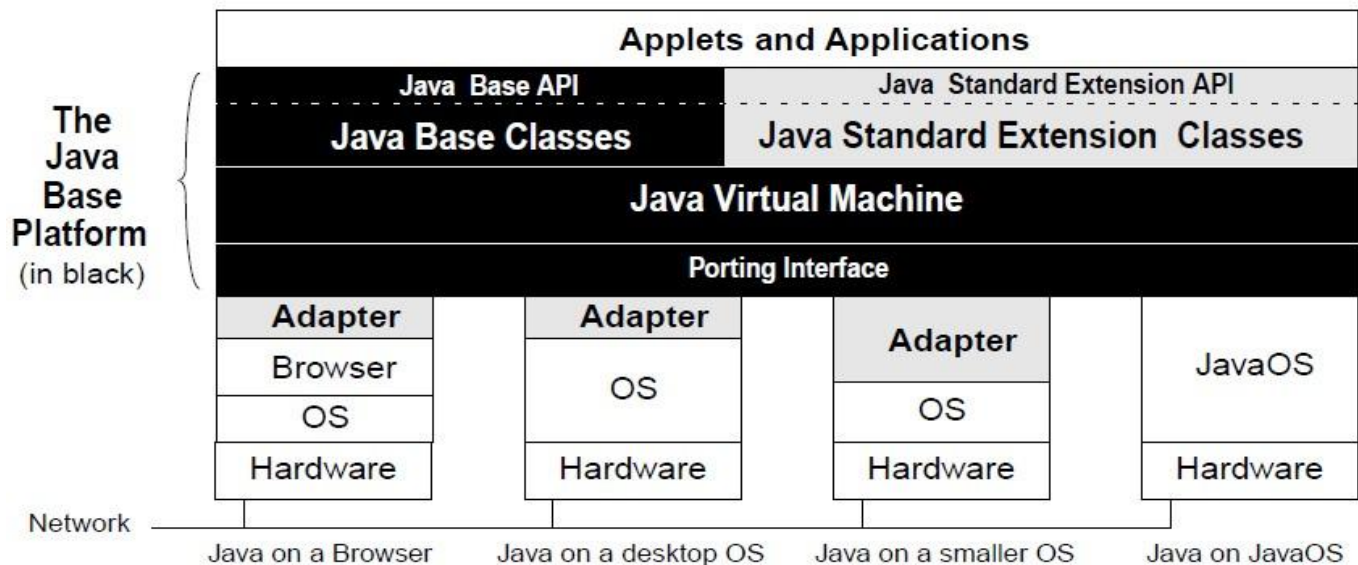
Developers use the Java Language to write source code for Java-powered applications. They compile once to the Java Platform, rather than to the underlying system. Java Language source code compiles to an intermediate, portable form of bytecodes that will run anywhere the Java Platform is present.

Developers can write object-oriented, multithreaded, dynamically linked applications using the Java Language. The platform has built-in security, exception handling, and automatic garbage collection. Just-in-time compilers are available to speed up execution by converting Java bytecodes into machine language. From within the Java Language, developers can also write and call native methods—methods in C, C++ or another language, compiled to a specific underlying operating system—for speed or special functionality.

The Java Language is the the entry ramp to the Java Platform. Programs written in the Java Language and then compiled will run on the Java Plaform.

The Java Platform has two basic parts:

- Java Virtual Machine
- Java Application Programming Interface (Java API)



Java Virtual Machine - The Java Virtual Machine is a “soft” computer that can be implemented in software or hardware. It’s an abstract machine designed to be implemented on top of existing processors. The porting interface and adapters enable it to be easily ported to new operating systems without being completely rewritten.

Java API - The Java API forms a standard interface to applets and applications, regardless of the underlying operating system. The Java API is the essential framework for application development.

This API specifies a set of essential interfaces in a growing number of key areas that developers will use to build their Java-powered applications.

The Java Base API provides the very basic language, utility, I/O, network, GUI, and applet services; OS companies that have licensed Java have contracted to include them in any Java Platform they deploy.

The Java Standard Extension API extends the capabilities of Java beyond the Java Base API. Some of these extensions will eventually migrate to the Java Base API. Other nonstandard extension APIs can be provided by the applet, application, or underlying operating system. As each new extension API specification is published, it will be made available for industry review and feedback before it is finalized.

A Java Language development environment includes both the compile-time and runtime environments, as shown in below figure. The Java Platform is represented by the runtime environment.

The developer writes Java Language source code (.java files) and compiles it to bytecodes (.class files). These bytecodes are instructions for the Java Virtual Machine.

To create an applet, the developer next stores these bytecode files on an HTTP server, and adds an <applet code=*filename*> tag to a Web page, which names the entry-point bytecode file.

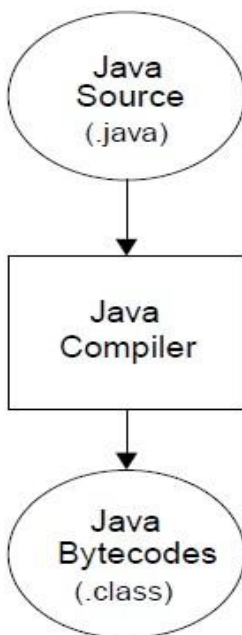
When an end user visits that page, the <applet> tag causes the bytecode files to be transported over the network from the server to the end user’s browser in the Java Platform. At this

end, the bytecodes are loaded into memory and then verified for security before they enter the Virtual Machine.

Once in the Virtual Machine, the bytecodes are interpreted by the Interpreter, or optionally turned into machine code by the just-in-time (JIT) code generator, known more commonly as the JIT Compiler.

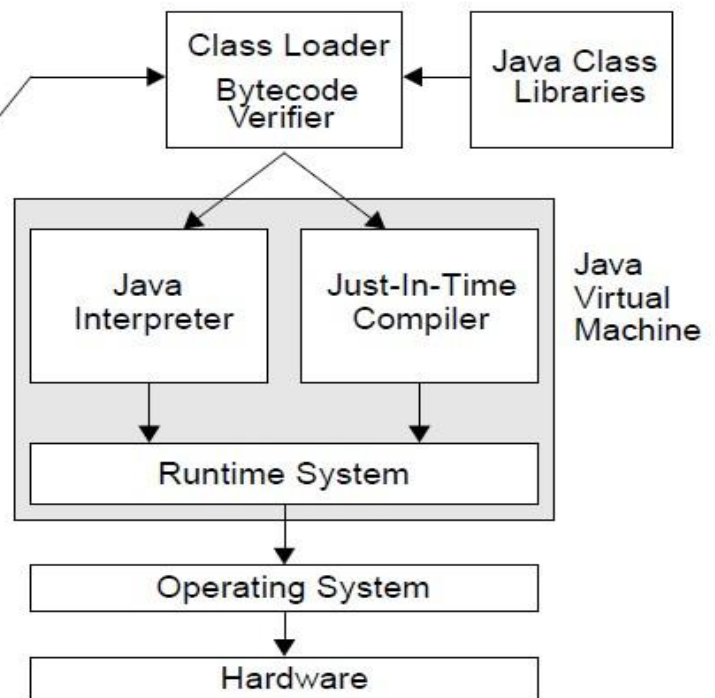
The Interpreter and JIT Compiler operate in the context of the runtime system (threads, memory, other system resources). Any classes from the Java Class Libraries (API) are dynamically loaded as needed by the applet.

Compile-time Environment



Java Bytecodes
move locally
or through
network

Runtime Environment (Java Platform)



3. EXPLAIN IN DETAIL ABOUT THE DATA TYPES IN JAVA

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:

- Primitive Data Types
- 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 keyword. Let us now look into detail about the eight primitive data types.

byte:

Byte data type is an 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: long 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 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");

Java Literals:

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.

Different Types of literals are:

- Number literals
- Character literals
- Boolean literals
- String literals

Number

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

For Example:

```
int decimal = 100;
```

```
int hexa = 0x64;
```

Boolean

Boolean literals consist of the keywords true and false

String and Character

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.

For Example:

```
“Hello World”
“two\nlines”
```

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:

<code>\n</code>	-	newline
<code>\t</code>	-	tab
<code>\”</code>	-	Double Quote
<code>\r</code>	-	Carriage Return
<code>\s</code>	-	Space

4. WRITE IN DETAIL ABOUT OPERATORS IN JAVA

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

Arithmetic Operators

Relational Operators

Bitwise Operators

Logical Operators

Assignment Operators

Misc Operators

THE ARITHMETIC OPERATORS:

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

Assume integer variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	A - B will give -10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200
/	Division - Divides left hand operand by right hand operand	B / A will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0
++	Increment - Increases the value of operand by 1	B++ gives 21
--	Decrement - Decreases the value of operand by 1	B-- gives 19

THE RELATIONAL OPERATORS:

There are following relational operators supported by Java language

Assume variable A holds 10 and variable B holds 20, then:

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

THE BITWISE OPERATORS:

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13; now in binary format they will be as follows:

a = 0011 1100

b = 0000 1101

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

The following table lists the bitwise operators:

Assume integer variable A holds 60 and variable B holds 13 then:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(A B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>>	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

The Logical Operators:

The following table lists the logical operators:

Assume Boolean variables A holds true and variable B holds false, then:

Operator	Description	Example
----------	-------------	---------

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

THE ASSIGNMENT OPERATORS:

There are following assignment operators supported by Java language:

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

&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
=	bitwise inclusive OR and assignment operator	C = 2 is same as C = C 2

MISC OPERATORS

There are few other operators supported by Java Language.

Conditional Operator (? :):

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:

```
variable x = (expression) ? value if true : value if false
```

Example:

```
public class Test
{
    public static void main(String args[])
    {
        int a , b;
        a = 10;
        b = (a == 1) ? 20: 30;
        System.out.println( "Value of b is : " + b );

        b = (a == 10) ? 20: 30;
        System.out.println( "Value of b is : " + b );
    }
}
```

This would produce the following result:

```
Value of b is : 30
Value of b is : 20
```

instanceof Operator:

This operator is used only for object reference variables. The operator checks whether the object is of a particular type(class type or interface type). instanceof operator is written as:

```
( Object reference variable ) instanceof ( class/interface type)
```

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is the example:

```
public class Test
{
    public static void main(String args[])
    {
        String name = "James";
        // following will return true since name is type of String
        boolean result = name instanceof String;
        System.out.println( result );
    }
}
```

This would produce the following result:

```
true
```

PRECEDENCE OF JAVA OPERATORS:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example, $x = 7 + 3 * 2$; here x is assigned 13, not 20 because operator $*$ has higher precedence than $+$, so it first gets multiplied with $3*2$ and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left to right
Unary	++ -- ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right

Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left
Comma	,	Left to right

5. EXPLAIN IN DETAIL ABOUT THE JAVA DECISION MAKING STATEMENTS IN JAVA

There are two types of decision making statements in Java.

They are:

- if statements
- switch statements

The if Statement:

An if statement consists of a Boolean expression followed by one or more statements.

Syntax:

The syntax of an if statement is:

```
if(Boolean_expression)
{
//Statements will execute if the Boolean expression is true
```

```
}
```

If the Boolean expression evaluates to true then the block of code inside the if statement will be executed. If not the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Example:

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

        if( x < 20 )
        {
            System.out.print("This is if statement");
        }
    }
}
```

This would produce the following result:

```
This is if statement
```

The if...else Statement:

An if statement can be followed by an optional *else* statement, which executes when the Boolean expression is false.

Syntax:

The syntax of an if...else is:

```
if(Boolean_expression)
{
    //Executes when the Boolean expression is true
}
else
{
    //Executes when the Boolean expression is false
}
```

Example:

```
public class Test {

    public static void main(String args[]){
        int x = 30;

        if( x < 20 ){
            System.out.print("This is if statement");
        }
    }
}
```

```
}else{
    System.out.print("This is else statement");
}
}
```

This would produce the following result:

```
This is else statement
```

The if...else if...else Statement:

An if statement can be followed by an optional *else if...else* statement, which is very useful to test various conditions using single if...else if statement.

When using if , else if , else statements there are few points to keep in mind.

An if can have zero or one else's and it must come after any else if's.

An if can have zero to many else if's and they must come before the else.

Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax:

The syntax of an if...else is:

```
if(Boolean_expression 1){
    //Executes when the Boolean expression 1 is true
}else if(Boolean_expression 2){
    //Executes when the Boolean expression 2 is true
}else if(Boolean_expression 3){
    //Executes when the Boolean expression 3 is true
}else {
    //Executes when the none of the above condition is true.
}
```

Example:

```
public class Test {

    public static void main(String args[]){
        int x = 30;

        if( x == 10 ){
            System.out.print("Value of X is 10");
        }else if( x ==
            20 ){ System.out.print("Value of X is
            20");
        }else if( x ==
            30 ){ System.out.print("Value of X is
            30");
        }else{
            System.out.print("This is else statement");
        }
    }
}
```

```
}
```

This would produce the following result:

```
Value of X is 30
```

Nested if...else Statement:

It is always legal to nest if-else statements which means you can use one if or else if statement inside another if or else if statement.

Syntax:

The syntax for a nested if...else is as follows:

```
if(Boolean_expression 1){  
    //Executes when the Boolean expression 1 is true  
    if(Boolean_expression 2){  
        //Executes when the Boolean expression 2 is true  
    }  
}
```

You can nest *else if...else* in the similar way as we have nested *if* statement.

Example:

```
public class Test {  
  
    public static void main(String args[]){  
        int x = 30;  
        int y = 10;  
  
        if( x == 30 ){  
            if( y == 10 ){  
                System.out.print("X = 30 and Y = 10");  
            }  
        }  
    }  
}
```

This would produce the following result:

```
X = 30 and Y = 10
```

THE SWITCH STATEMENT:

A *switch* statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax:

The syntax of enhanced for loop is:

```

switch(expression){
  case value :
    //Statements
    break; //optional
  case value :
    //Statements
    break; //optional
  //You can have any number of case statements.
  default : //Optional
    //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.

Example:

```

public class Test {

  public static void main(String args[]){
    //char grade = args[0].charAt(0);
    char grade = 'C';

    switch(grade)
    {
      case 'A' :
        System.out.println("Excellent!");
        break;
      case 'B' :
      case 'C' :
        System.out.println("Well done");
        break;
      case 'D' :
        System.out.println("You passed");
      case 'F' :
        System.out.println("Better try again");
        break;
    }
  }
}

```



```

    default :
        System.out.println("Invalid grade");
    }
    System.out.println("Your grade is " + grade);
}
}

```

Compile and run above program using various command line arguments. This would produce the following result:

```

$ java Test
Well done
Your grade is C
$

```

6. EXPLAIN IN DETAIL ABOUT THE LOOPING STATEMENTS IN JAVA

There may be a situation when we need to execute a block of code several number of times, and is often referred to as a loop.

Java has very flexible three looping mechanisms. You can use one of the following three loops:

- while Loop
- do...while Loop
- for Loop

As of Java 5, the *enhanced for loop* was introduced. This is mainly used for Arrays.

The while Loop:

A while loop is a control structure that allows you to repeat a task a certain number of times.

Syntax:

The syntax of a while loop is:

```

while(Boolean_expression)
{
    //Statements
}

```

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

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

Example:

```

public class Test {

```

```

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

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

```

This would produce the following result:

```

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

```

The do...while Loop:

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

Syntax:

The syntax of a do...while loop is:

```

do
{
    //Statements
}while(Boolean_expression);

```

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

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

Example:

```

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

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

```

```
}  
}
```

This would produce the following result:

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

The for Loop:

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

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

Syntax:

The syntax of a for loop is:

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

Here is the flow of control in a for loop:

The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

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.

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.

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.

Example:

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

```
}
```

This would produce the following result:

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

Enhanced for loop in Java:

As of Java 5, the enhanced for loop was introduced. This is mainly used for 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 evaluates 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.o  
            ut.print( x );  
            System.out.print(",");  
        }  
        System.out.print("\n");  
        String [] names ={"James", "Larry", "Tom", "Lacy"};  
        for( String name : names )  
            { System.out.print( name )  
            ; System.out.print(",");  
            }  
        }  
}
```

This would produce the following result:

10,20,30,40,50,
James,Larry,Tom,Lacy,

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");  
        }  
    }  
}
```

This would produce the following result:

```
10  
20
```

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;
```

Example:

```
public class Test {
```

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

This would produce the following result:

```
10
20
40
50
```

7. WRITE IN DETAIL ABOUT CLASSES AND OBJECTS IN JAVA

SYNOPSIS

- Object
- Class
- Class Model
- Access Specifiers
- Variables
- Methods
- Object Creation
- Accessing Members of object
- Example Program

OBJECT:

An object is an real world entity that has its own properties (or) state and actions (or) behavior

EX:

a pen, a person, a student

CLASS:

A class is a template/blue print of objects those have similar properties (or) state and actions (or) behavior

Ex:

Pen, Person, Student

CLASS MODEL:

```
<access-specifiers> class class_name
{
    access-specifiers data-type variable-name1;
    access-specifiers data-type variable-name2;
    access-specifiers return-type method_name1(arg1,arg2,arg3)
    {
        .....
        .....
    }
    access-specifiers return-type method_name2(arg1)
    {
        .....
        .....
    }
}
```

EX:

```
public class Sample
{
    private int a;
    public double average;
    double total;
    public void add()
    {
        System.out.println("METHOD 1");
    }
    void display(int k, double j)
    {
        System.out.println("METHOD 2");
    }
}
```

ACCESS SPECIFIERS:

Access specifiers specifies the type of access levels for the classes, variables, methods and constructor

Types of access-specifiers:

Default - Visible to the package, the default.

Private - Visible to the class only.

Public - Visible to the world.

Protected -Visible to the package and all subclasses.

VARIABLES:

Variable is name of reserved area allocated in memory. Ex: `int data=50; //data is variable`

Types of variable:

Local Variable:

A variable that is declared inside the method is called local variable

Instance Variable:

A variable that is declared inside the class but outside the method is called instance variable. It is not declared as static

Static Variable:

A variable that is declared as static is called static variable. It is common to all objects

Ex:

```
class A
{
    int data=50;//instance variable
    static int m=100;//static variable
    void method()
    {
        int n=90; //local variable
    }
} //end of class
```

METHODS:

A Java method is a collection of statements that are grouped together to perform an operation
Methods are also known as Procedures or Functions

Syntax:

Method definition consists of a method header and a method body

```
access-specifier returnType nameOfMethod (Parameter List)
{
    // method body
```



```
}  
Ex:  
    public void add(int a, int b)  
    {  
        System.out.println(a+b);  
    }
```

OBJECT CREATION:

Syntax:

```
class_name object_name=new class_name();
```

It has two parts:

object declaration

Associate an object name with its class type to reserve proper amount of memory for that object

```
class_name object_name; --> Student s1;
```

object instantiation

Creation of object using new operator in the heap memory and return the reference to the stack memory

```
object_name=new class_name(); --> s1=new Student();
```

THE NEW OPERATOR:

The new operator creates a single instance of a named class and returns a reference to that object.

Ex:

```
point p = new point();
```

p is a reference of the new point object.

ACCESSING OF OBJECT MEMBERS:

The dot (.) operator is used to access the instance variables and methods withing an object.

General Form:

```
Object reference (.) Variable name
```

Object Reference is a reference to an object and variable name is the name of the instance.

Ex:

```
p.x=10;
```

```
p.y=20;
```

EXAMPLE PROGRAM:

```
import java.util.*;
```

```

class Student
{
    private int regNo,m1,m2,m3,total;
    private String name,dept,res;
    Scanner s=new Scanner(System.in);

    public void getDetails()
    {
        System.out.println("Enter Register No.");
        regNo=s.nextInt();
        System.out.println("Enter Name:");
        name=s.next();
        System.out.println("Enter Department");
        dept=s.next();
        System.out.println("Enter Mark1");
        m1=s.nextInt();
        System.out.println("Enter Mark2");
        m2=s.nextInt();
        System.out.println("Enter Mark3");
        m3=s.nextInt();
    }

    public void display()
    {
        System.out.println("Name:" + name + "Reg No." + regNo );
        System.out.println("Mark 1:" + m1 + "Mark 2:" + m2 + " Mark 3:" + m3);
        System.out.println("total:"+total+"result:"+res);
    }
    public void cal()
    {
        total=m1+m2+m3;
        if((m1>=50) && (m2>=50) && (m3>=50))
        {
            res="pass";
        }
        else
        {
            res="fail";
        }
    }

    public static void main(String arg[])
    {

```

```

        Student s1=new Student();
        s1.getDetails();
        s1.cal();
        s1.display();
    }
}

```

8. WRITE SHORT NOTES ABOUT CONSTRUCTORS AND DESTRUCTORS

CONSTRUCTORS:

Constructor in java is a special type of method that is used to initialize the object. Java constructor is invoked at the time of object creation.

It constructs the values i.e. provides data for the object that is why it is known as constructor

CHARACTERISTICS OF CONSTRUCTOR:

Constructor is used to initialize object.

Constructor must not have return type.

Constructor is invoked implicitly.

The java compiler provides a default constructor if you don't have any constructor.

Constructor name must be same as the class name.

Generally constructor is declared as public mode.

Constructor can be overloaded.

Constructor may be virtual.

Constructor of one class can call constructor of base class using **super** keyword.

TYPES OF CONSTRUCTOR:

Default Constructor

Parameterized Constructor

Default constructor:

It is also known as no-arg constructor. Constructor with no arguments is known as default constructor.

Syntax:

```

<class_name>()
{
    .....
}

```

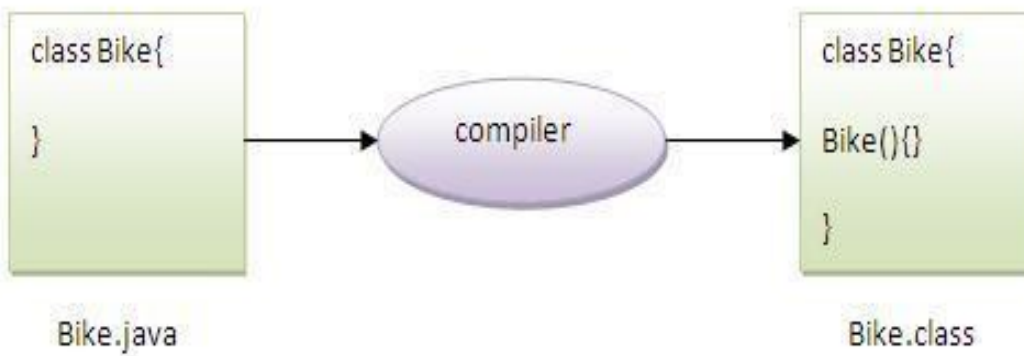
Ex:

```
class Car
{
    public Car()
    {
        System.out.println("Car created");
    }
    public static void main(String arg[])
    {
        Car c= new Car();
    }
}
```

OUTPUT:

Car created

If there is no constructor in a class, compiler automatically creates a default constructor



Parameterized constructor:

Constructor with argument list is known as parameterized constructor.
Parameterized constructor is used to provide different values to the distinct objects

Syntax:

```
<class_name>(type arg1, type arg2)
{
    .....
}
```

EXAMPLE:

```
class Student
{
    int id;
    String name;

    Student(int i,String n)
    {
        id = i;
        name = n;
    }
    void display()
    {
        System.out.println(id+" "+name);
    }

    public static void main(String args[])
    {
        Student s1 = new Student(101,"Guru");
        Student s2 = new Student(102,"Mani");
        s1.display();
        s2.display();
    }
}
```

OUTPUT:

101 Guru

102 Mani

Destructor and finalize() method

A destructor is a special method typically used to perform cleanup after an object is no longer needed by the program. C++ supports destructors, but JAVA does not support destructors.

JAVA supports another mechanism for returning memory to the operating system when it is no longer needed by an object.

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.

Example of finalize() method:

```
public class Thing {  
  
    public static int number_of_things = 0;  
    public String what;  
  
    public Thing (String what)  
        { this.what = what;  
          number_of_things++;  
        }  
  
    protected void finalize () { //Destructor function  
        number_of_things--;  
    }  
}
```

Garbage Collection

Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator.

Java takes a different approach; It handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection*.

It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++.

Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used.

Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

9. EXPLAIN CONSTRUCTORS IN JAVA WITH AN EXAMPLE PROGRAM

SYNOPSIS

CONSTRUCTORS
CHARACTERISTICS OF CONSTRUCTORS
TYPES OF CONSTRUCTORS
DEFAULT CONSTRUCTOR WITH SYNTAX AND EXAMPLE
PARAMETRISED CONSTRUCTOR WITH SYNTAX AND EXAMPLE

CONSTRUCTOR OVERLOADING
SAMPLE PROGRAM
DIFFERENCE BETWEEN CONSTRUCTOR AND METHOD

CONSTRUCTORS:

Constructor in java is a special type of method that is used to initialize the object. Java constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object that is why it is known as constructor

CHARACTERISTICS OF CONSTRUCTOR:

Constructor is used to initialize object.
Constructor must not have return type.
Constructor is invoked implicitly.
The java compiler provides a default constructor if you don't have any constructor.
Constructor name must be same as the class name.
Generally constructor is declared as public mode.
Constructor can be overloaded.
Constructor may be virtual.
Constructor of one class can call constructor of base class using **super** keyword.

TYPES OF CONSTRUCTOR:

Default Constructor
Parameterized Constructor

Default constructor:

It is also known as no-arg constructor. Constructor with no arguments is known as default constructor.

Syntax:

```
<class_name>()  
{  
    .....  
}
```

Ex:

```
class Car  
{
```

```

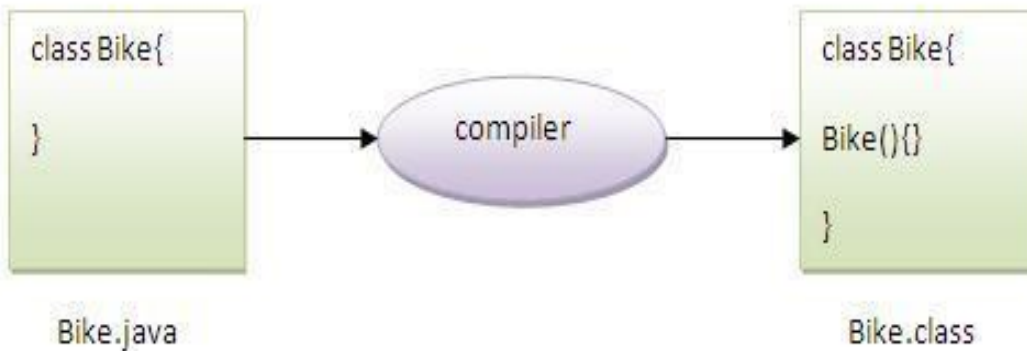
public Car()
{
    System.out.println("Car created");
}
public static void main(String arg[])
{
    Car c= new Car();
}
}

```

OUTPUT:

Car created

If there is no constructor in a class, compiler automatically creates a default constructor



Parameterized constructor:

Constructor with argument list is known as parameterized constructor.
Parameterized constructor is used to provide different values to the distinct objects

Syntax:

```

<class_name>(type arg1, type arg2)
{
    .....
}

```

EXAMPLE:

```

class Student

```



```

{
    int id;
    String name;

    Student(int i,String n)
    {
        id = i;
        name = n;
    }
    void display()
    {
        System.out.println(id+" "+name);
    }

    public static void main(String args[])
    {
        Student s1 = new Student(101,"Guru");
        Student s2 = new Student(102,"Mani");
        s1.display();
        s2.display();
    }
}

```

OUTPUT:

101 Guru

102 Mani

CONSTRUCTOR OVERLOADING

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists

The compiler differentiates these constructors in two ways
the different number of parameters
the different type of parameters

EXAMPLE:

```

class Student
{
    int id, age; String
    name; Student(int
    i,String n)
    {
        id = i;

```

```

        name = n;
    }
    Student(int i,String n,int a)
    {
        id = i;
        name = n;
        age = a;
    }
    void display()
    {
        System.out.println(id+" "+name+" "+age);
    }

    public static void main(String args[])
    {
        Student s1 = new Student(101,"Guru");
        Student s2 = new Student(102,"Mani",30);
        s1.display();
        s2.display();
    }
}

```

OUTPUT:

101 Guru 0

102 Mani 30

DIFFERENCE BETWEEN CONSTRUCTOR AND METHODS

Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.

Constructor name must be same as the class name.

Method name may or may not be same as class name.

10. EXPLAIN IN DETAIL ABOUT PACKAGES IN JAVA SYNOPSIS

INTRODUCTION
BENEFITS
TYPES OF PACKAGES
ACCESSING CLASS FROM A PACKAGE
CREATING YOUR OWN PACKAGE
DEFAULT PACKAGE
FINDING PACKAGE
CLASSPATH ENVIRONMENT VARIABLE
ACCESS SPECIFIERS
EXAMPLE

INTRODUCTION:

Packages provide a mechanism for grouping a variety of classes and / or interfaces together.

Grouping is based on functionality

Benefits of Packages:

The classes contained in the packages of other programs can be reused.

In packages, classes can be unique compared with classes in other packages.

Packages provides a way to hide classes

Types of packages:

Pre-defined packages or Java API packages

User-defined packages

Java API packages:

A large number of classes grouped into different packages based on functionality.

Examples:

1. java.lang
2. java.util
3. java.io
4. java.awt
5. java.net
6. java. applet etc.

Accessing Classes in a Package

1. Fully Qualified class name:
Example: java.awt.Color

2. `import packagename.classname;`
Example: `import java.awt.Color;`
or
`import packagename.*;`
Example: `import java.awt.*;`

Import statement must appear at the top of the file, before any class declaration

Creating Your Own Package

1. Declare the package at the beginning of a file using the form
`package packagename;`
2. Define the class that is to be put in the package and declare it public.
3. Create a subdirectory with same name as package name under the directory where the main source files are stored.
4. Store the listing as `classname.java` in the subdirectory created.
5. Compile the file. This creates `.class` file in the subdirectory

Syntax:

```
package [PackageName];
public class [ClassName]
{
    //Body of the class
}
```

Example:

```
package firstPackage;
public class FirstClass
{
    //Body of the class
}
```

Example:

```
//Class in package
package p1;
public class ClassA
{
    public void displayA( )
    {
        System.out.println("Class A");
    }
}
```

```

//Importing class from package
import p1.*;
class testclass
{
    public static void main(String str[])
    {
        ClassA obA=new ClassA();
        obA.displayA();
    }
}

```

Creating Packages:

Consider the following declaration:

```
package firstPackage.secondPackage;
```

This package is stored in subdirectory named **firstPackage.secondPackage**.

A java package can contain more than one class definitions that can be declared as public.

Only one of the classes may be declared **public** and that class name with **.java** extension is the source file name.

Default Package:

If a source file does not begin with the *package* statement, the classes contained in the source file reside in the *default package*

The java compiler automatically looks in the default package to find classes

Finding Packages:

1. By default, java runtime system uses current directory as starting point and search all the subdirectories for the package.
2. Specify a directory path using CLASSPATH environmental variable

CLASSPATH Environment Variable

The compiler and runtime interpreter know how to find standard packages such as *java.lang* and *java.util*

The CLASSPATH environment variable is used to direct the compiler and interpreter to where programmer defined imported packages can be found

The CLASSPATH environment variable is an ordered list of directories and files

To set the CLASSPATH variable we use the following command:

```
set CLASSPATH=c:\
```

Java compiler and interpreter searches the user defined packages from the above directory.

To clear the previous setting we use:

```
set CLASSPATH=
```

ACCESS SPECIFIERS:

Access specifiers specifies the type of access levels for the classes, variables, methods and constructor

Types of access-specifiers:

Default - Visible to the package, the default.

Private - Visible to the class only.

Public - Visible to the world.

Protected - Visible to the package and all subclasses.

Levels of Access Control:

	public	protected	friendly (default)	private
same class	Yes	Yes	Yes	Yes
Subclass in the same package	Yes	Yes	Yes	No
Other class in the same package	Yes	Yes	Yes	No
Subclass in other packages	Yes	Yes	No	No
Non-subclass in other package	Yes	No	No	No

Example 1:

```
package my_package;
class A // package scope
{
    // A's public & private members
}
public class B // public scope
{
    // B's public and private members
}
```

Example 2:

```

package my_package;
class A
{
    int get() { return data; }           // package scope
    public A(int d) { data=d;}         // public scope
    private int data;                  // private scope
}
class B
{
    void f()
    {
        A a=new A(d);                 // OK A has package scope
        int d=a.get();                 // OK – get() has package scope
        int d1=a.data;                 // Error! – data is private
    }
}

```

11. WRITE IN DETAIL ABOUT INTERFACES IN JAVA

INTERFACES

An **interface in java** is a blueprint of a class. It has static constants and abstract methods only.

The interface in java is a **mechanism to achieve fully abstraction**. There can be only abstract methods in the java interface not method body.

It is used to achieve fully abstraction and multiple inheritance in Java.

Java Interface also **represents IS-A relationship**.

It cannot be instantiated just like abstract class.

USE OF INTERFACE:

There are mainly three reasons to use interface. They are given below.

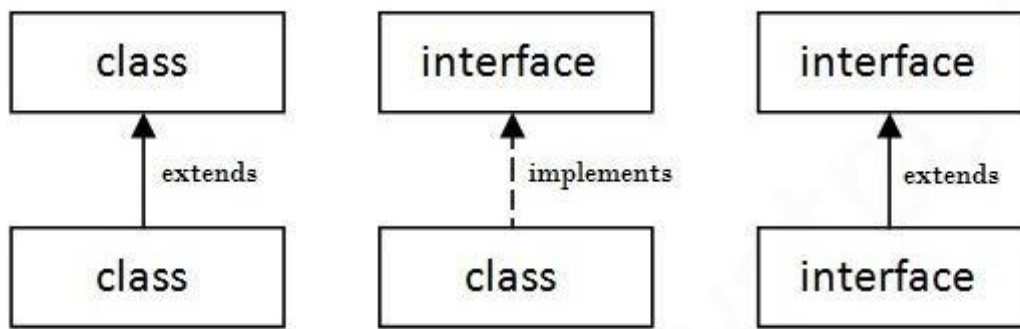
It is used to achieve fully abstraction.

By interface, we can support the functionality of multiple inheritance.

It can be used to achieve loose coupling.

RELATIONSHIP BETWEEN CLASSES AND INTERFACES

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**



DIFFERENCE BETWEEN CLASSES AND INTERFACES

CLASS	INTERFACE
The members of a class can be constant or variables.	The members of an interface are always declared as constant , i.e., their values are final.
The class definition can contain the code for each of its methods. That is , the methods can be abstract or non-abstract.	The methods in an interface are abstract in nature , i.e., there is no code associated with them. It is later defined by the class that implements the interface.
It can be instantiated by declaring objects.	It cannot be used to declare objects. It can only be inherited by a class.
It can use various access specifiers like public , private or protected.	It can only use the public access specifier.

INTERFACE SYNTAX:

```

interface interfaceName
{
    Variables declaration;
    Method declaration;
}
  
```

interface - keyword.
interfacename - any valid java variable.

SIMPLE EXAMPLE OF JAVA INTERFACE

In this example, Printable interface have only one method, its implementation is provided in the A class.

```

interface printable{
void print();
}
  
```



```

class A6 implements printable{
public void print(){System.out.println("Hello");}

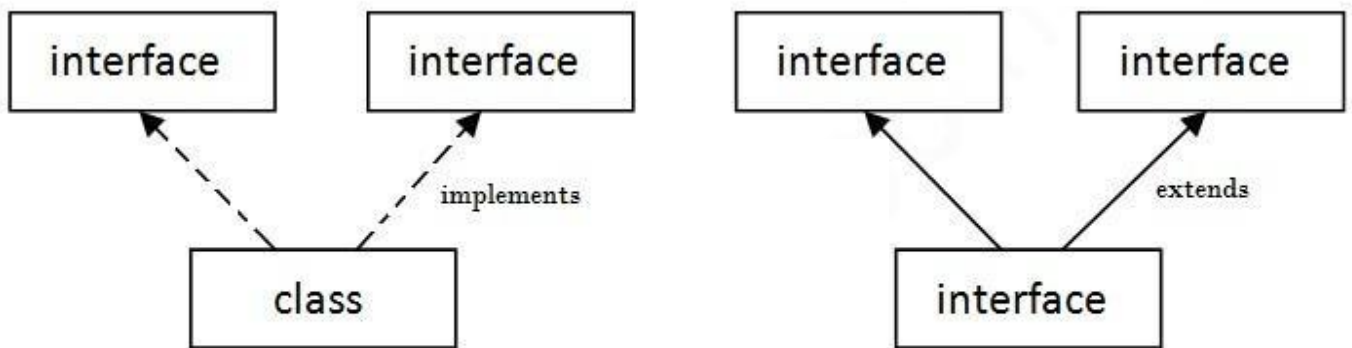
public static void main(String
args[]){ A6 obj = new A6();
obj.print();
}
}

```

Output: Hello

MULTIPLE INHERITANCE IN JAVA BY INTERFACE

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Multiple Inheritance in Java

```

interface Printable{
void print();
}

```

```

interface Showable{
void show();
}

```

```

class A7 implements Printable,Showable{

public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String
args[]){ A7 obj = new A7();
obj.print();
}
}

```

```
obj.show();  
}  
}
```

Output:Hello
Welcome

12. WRITE SHORT NOTES ON INTERNALIZATION IN JAVA

INTERNATIONALIZATION AND LOCALIZATION IN JAVA

- **Internationalization** is also abbreviated as I18N because there are total 18 characters between the first letter 'I' and the last letter 'N'.
- Internationalization is a mechanism to create such an application that can be adapted to different languages and regions.
- Internationalization is one of the powerful concept of java if you are developing an application and want to display messages, currencies, date, time etc. according to the specific region or language.
- **Localization** is also abbreviated as I10N because there are total 10 characters between the first letter 'L' and last letter 'N'.
- Localization is the mechanism to create such an application that can be adapted to a specific language and region by adding locale-specific text and component.

Understanding the culturally dependent data before starting internationalization

Before starting the internationalization, Let's first understand what are the informations that differ from one region to another.

There is the list of culturally dependent data:

Messages

Dates

Times

Numbers

Currencies

Measurements

Phone Numbers

Postal Addresses

Labels on GUI components etc.

Importance of Locale class in Internationalization

An object of Locale class represents a geographical or cultural region. This object can be used to get the locale specific information such as country name, language, variant etc.

Fields of Locale class

There are fields of Locale class:

1. public static final Locale ENGLISH
2. public static final Locale FRENCH
3. public static final Locale GERMAN
4. public static final Locale ITALIAN
5. public static final Locale JAPANESE
6. public static final Locale KOREAN
7. public static final Locale CHINESE
8. public static final Locale SIMPLIFIED_CHINESE
9. public static final Locale TRADITIONAL_CHINESE
10. public static final Locale FRANCE
11. public static final Locale GERMANY
12. public static final Locale ITALY
13. public static final Locale JAPAN
14. public static final Locale KOREA
15. public static final Locale CHINA
16. public static final Locale PRC
17. public static final Locale TAIWAN
18. public static final Locale UK
19. public static final Locale US
20. public static final Locale CANADA
21. public static final Locale CANADA_FRENCH
22. public static final Locale ROOT

Constructors of Locale class

There are three constructors of Locale class. They are as follows:

1. Locale(String language)
2. Locale(String language, String country)
3. Locale(String language, String country, String variant)

Commonly used methods of Locale class

There are given commonly used methods of Locale class.

1. **public static Locale getDefault()** it returns the instance of current Locale
2. **public static Locale[] getAvailableLocales()** it returns an array of available locales.
3. **public String getDisplayCountry()** it returns the country name of this locale object.
4. **public String getDisplayLanguage()** it returns the language name of this locale object.
5. **public String getDisplayVariant()** it returns the variant code for this locale object.
6. **public String getISO3Country()** it returns the three letter abbreviation for the current locale's country.
7. **public String getISO3Language()** it returns the three letter abbreviation for the current locale's language.

Example of Local class that prints the informations of the default locale

In this example, we are displaying the informations of the default locale. If you want to get the informations about any specific locale, comment the first line statement and uncomment the second line statement in the main method.

```
import java.util.*;
public class LocaleExample {
public static void main(String[] args)
{ Locale locale=Locale.getDefault();
//Locale locale=new Locale("fr","fr");//for the specific locale

System.out.println(locale.getDisplayCountry());
System.out.println(locale.getDisplayLanguage());
System.out.println(locale.getDisplayName());
System.out.println(locale.getISO3Country());
System.out.println(locale.getISO3Language());
System.out.println(locale.getLanguage());
System.out.println(locale.getCountry());

}
}
```

Output:

```
United States
English
English (United States)
USA
eng
en
US
```

UNIT 2 - 2 MARKS**1. WHAT IS METHOD OVERRIDING?**

If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in java

This is possible by defining a method in the subclass that has the *same name*, *same arguments* and *same return type* as a method in the superclass

When the method is called, the method defined in the subclass is invoked and executed instead of the one in the superclass

2. WHAT IS THE USE OF METHOD OVERRIDING?

to provide specific implementation of a method that is already provided by its super class
runtime polymorphism

3. WRITE DOWN THE RULES OF METHOD OVERRIDING?

method must have same name as in the parent class

method must have same parameter as in the parent class.

must be IS-A relationship (inheritance)

4. WHAT IS METHOD OVERLOADING AND METHOD OVERRIDING?

When a method in a class having the same method name with different arguments is said to be method overloading. Method overriding: When a method in a class having the same method name with same arguments is said to be method overriding

5. DEFINE FINAL MODIFIER?

All methods and variables can be overridden by default in subclasses.

To prevent the subclasses from overriding the members of the superclass, we can declare them as final by using the keyword ***final*** as a modifier.

Example:

```
final int SIZE = 100;
final void showstatus ()
{
.....
}
```

The functionality defined in this method will never be altered in any way

6. DEFINE INHERITANCE

The mechanism of deriving a new from a from an old one is called inheritance

The old class is known as the ***base class*** or ***super class*** or ***parent class***

The new class is called the ***subclass*** or ***derived class*** or ***child class***

The inheritance allows subclasses to inherit all the variables and methods of their parent classes

7. STATE THE USES OF INHERITANCE

Code Reusability

Method Overriding (so runtime polymorphism can be achieved).

8. STATE THE TYPES OF INHERITANCE

Single inheritance [only one super class]

Hierarchical inheritance [one super classes, many subclasses]

Multilevel inheritance [derived from a derived class].

9. WHY MULTIPLE INHERITANCE IS NOT SUPPORTED IN JAVA

To reduce the complexity and simplify the language
To remove ambiguity

10. SYNTAX TO EXTEND A CLASS

```
class subclassname extends superclassname
{
    Variable declaration;
    Methods declaration;
}
```

The keyword **extends** signifies that the properties of the **superclassname** are extended to the **subclassname**

11. DEFINE SUPER KEYWORD

The **super** keyword in java is a reference variable that is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable

12. WHAT IS THE USE OF SUPER KEYWORD

super is used to refer immediate parent class instance variable
super() is used to invoke immediate parent class constructor
super is used to invoke immediate parent class method

13. WHAT IS A STREAM?

A stream is a sequence of data. In Java a stream is composed of bytes. It's called a stream because it's like a stream of water that continues to flow

14. WHAT ARE THE THREE STREAM THAT AUTOMATICALLY GENERATED IN JAVA

- System.out: standard output stream
- System.in: standard input stream
- System.err: standard error stream

EX:

```
System.out.println("simple message");
System.err.println("error message");
```

EX:

```
int i=System.in.read();//returns ASCII code of 1st character
```

15. WHAT ARE THE COMMONLY USED METHODS IN OUTPUTSTREAM CLASS

Method	Description
1) public void write(int) throws IOException:	is used to write a byte to the current output stream.
2) public void write(byte[]) throws IOException:	is used to write an array of byte to the current output stream.
3) public void flush() throws IOException:	flushes the current output stream.
4) public void close() throws IOException:	is used to close the current output stream.

16.WHAT ARE THE COMMONLY USED METHODS IN INPUTSTREAM CLASS

Method	Description
1) public abstract int read()throws IOException:	reads the next byte of data from the input stream.It returns -1 at the end of file.
2) public int available()throws IOException:	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException:	is used to close the current input stream.

17.WHAT ARE THE VARIOUS CLASSES USED IN FILE HANDLING

FileOutputStream – to write byte data to a file

FileInputStream – to read byte data from file

FileReader - to read character oriented data from file

FileWriter -to write character oriented data to a file

18.HOW COULD JAVA CLASSES DIRECT PROGRAM MESSAGES TO THE SYSTEM CONSOLE, BUT ERROR MESSAGES, SAY TO A FILE?

The class System has a variable out that represents the standard output, and the variable err that represents the standard error device. By default, they both point at the system console.

This how the standard output could be re-directed:

```
Stream st = new Stream(new FileOutputStream("output.txt")); System.setErr(st);
System.setOut(st);
```

19.WHAT IS MULTI-THREADING

A process is a program in execution. Two or more processes running concurrently in a computer is called multi-tasking. In java it is term as **multi-threading**.

Ex. While typing a document, we can take printout.

20.WHAT IS THREAD

A single sequential flow of control.

All programs have at least one thread-the main thread.

More than one thread can executing concurrently.

21.WHAT ARE THE ADVANTAGES OF MULTITHREADING

Reduces complexity of large programs.

Maximizes CPU utilization.

Increases the speed of execution.

22.WHAT ARE ALL THE METHODS USED IN THREAD

Java's multi-threading operation is based on the thread class. Thread is a class and it contains constructors and methods for creating threads. Thread class is available in **java.lang** package. Thread class contains different methods to control threads. They are,

Run()
 Start()
 Yield()
 Sleep()
 Stop()
 Suspend()
 Resume()
 Notify()

23.WHAT'S THE DIFFERENCE BETWEEN THE METHODS SLEEP() AND WAIT()

The code `sleep(1000);` puts thread aside for exactly one second. The code `wait(1000)`, causes a wait of up to one second. A thread could stop waiting earlier if it receives the `notify()` or `notifyAll()` call. The method `wait()` is defined in the class `Object` and the method `sleep()` is defined in the class `Thread`

24.HOW TO DEFINE AND RUN A THREAD

A thread class can be created by extending the `Thread` class. The `Thread` class is available in `java.lang` package. The following steps are followed to define and run a thread.

Declare the class by extending **Thread** class.
 Override the **run()** method in extended class.
 Create the thread object using the class declared.
 Call the **start()** method to initiate **Thread** execution.

25.WHAT IS MEAN BY LIFE CYCLE OF A THREAD

There are five states in a lifetime of a thread. at a time, the thread may be in any one of the following five states. They are,

New born state
 Runnable state
 Running state
 Blocked state
 Dead state

26.DEFINE THREAD PRIORITIES.

The programmer can set the priorities for threads. This can be done using **setpriority()** method of `Thread` class. The syntax is,

Threadname.setpriority (value);

The value is an integer value. This integer value must be between 1 to 10. In addition the `Thread` class defines some predefined constants also.

`MIN_PRIORITY` = 1
`NORM_PRIORITY` = 5
`MAX_PRIORITY` = 10

The default value is 5.

27. DEFINE THREAD SCHEDULING

Allocating CPU time for the threads in the program is called thread scheduling.

28. WHAT IS MEAN BY THREAD SYNCHRONIZATION

The data and the methods that are common to many threads may commonly be placed outside the thread.

In such situations more than one thread can try to access the same method and data at the same time or a thread can try to access the method that is currently in use by another thread.

This problem can be overcome by using a technique called synchronization the general form is,

```
Synchronized return-type method-name
{
    -----
    Statements;
    -----
}
```

29. WHICH CLASS IS THE WAIT() METHOD DEFINED IN?

The wait() method is defined in the Object class, which is the ultimate superclass of all others

30. WHAT IS A GREEN THREAD?

A green thread refers to a mode of operation for the Java Virtual Machine (JVM) in which all code is executed in a single operating system thread.

31. WHAT ARE NATIVE OPERATING SYSTEM THREADS?

Native operating system threads are those provided by the computer operating system that plays host to a Java application, be it Windows, Mac or GNU/Linux.

32. WHAT IS EXCEPTION

An exception is a condition caused by runtime error in the program. Java interpreter encounters an error such as division by zero and throws it

33. WHAT IS THE PURPOSE OF EXCEPTION HANDLING

Exception handling mechanism is to provide a means to detect and report an "exceptional circumstance".

This mechanism suggests a separate error handling code.

34. WHAT ARE THE SEGMENTS IN EXCEPTION HANDLING MECHANISM

Find the problem(Hit the exception)

Inform that an error has occurred(throw the exception)

Receive the error information(catch the exception)

Take corrective measures(Handle the exception)

35. WRITE DOWN THE SYNTAX FOR EXCEPTION HANDLING

```
try
{
    Statemen
t;
}
catch(exception_type e)
{
    Statement;
}
```

36. DEFINE THE TWO BLOCKS IN EXCEPTION HANDLING

Try Block – Statement causing exception

Catch block – Statement handling the exception

37. MENTION SOME OF THE TYPES OF EXCEPTION

Arithmetic exception

Arrayindexoutof bound exception

Arrayshareexception

ClassCastException

NegativeArraysizeException

SecurityException

38. DOES IT MATTER IN WHAT ORDER CATCH STATEMENTS FOR FILENOTFOUNDEXCEPTION AND IOEXCEPTIPON ARE WRITTEN?

Yes, it does. The FileNotFoundException is inherited from the IOException. Exception's subclasses have to be caught first

39. STATE THE KEYWORDS IN EXCEPTION HANDLING

There are 5 keywords used in java exception handling.

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

40. DEFINE JAVA FINALLY BLOCK

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block must be followed by try or catch block

41. WHAT'S THE DIFFERENCE BETWEEN AN INTERFACE AND AN ABSTRACT CLASS?

An abstract class may contain code in method bodies, which is not allowed in an interface. With abstract classes, you have to inherit your class from it and Java does not allow multiple inheritance. On the other hand, you can implement multiple interfaces in your class

42. DEFINE ABSTRACT CLASS?

Abstract classes are classes from which instances are usually not created. It is basically used to contain common characteristics of its derived classes. Abstract classes are generally higher up the hierarchy and act as super classes. Methods can also be declared as abstract. This implies that non-abstract classes must implement these methods

43. WHAT'S THE DIFFERENCE BETWEEN AN INTERFACE AND AN ABSTRACT CLASS?

An abstract class may contain code in method bodies, which is not allowed in an interface. With abstract classes, you have to inherit your class from it and Java does not allow multiple inheritance. On the other hand, you can implement multiple interfaces in your class

11 MARKS**1. WRITE IN DETAIL ABOUT METHOD OVERLOADING WITH EXAMPLE PROGRAM****SYNOPSIS**

METHOD OVERLOADING
 DIFFERENT WAYS TO OVERLOAD THE METHOD
 DIFFERENT NO. OF ARGUMENTS
 DIFFERENT TYPES OF ARGUMENTS
 OVERLOADING MAIN METHOD
 METHOD OVERLOADING AND TYPEPROMOTION

METHOD OVERLOADING:

- If a class have multiple methods by same name but different parameters, it is known as method overloading.
- If we have to perform only one operation having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs. So, we perform method overloading to figure out the program quickly

ADVANTAGE OF METHOD OVERLOADING

increases the readability of the program

DIFFERENT WAYS TO OVERLOAD THE METHOD :

There are two ways to overload the method in java

- * By changing the number of arguments
- * By changing the data type

In java, method overloading is not possible by changing the return type of the method.

EXAMPLE PROGRAMS:**1) DIFFERENT NO.OF ARGUMENTS :**

```
class calculation
{
    void sum(int a,int b)
    {
        System.out.println(a+b);
    }
    void sum(int a,int b,int c)
    {
```

```

        System.out.println(a+b+c);
    }
    public static void main(String args[])
    {
        Calculation obj=new calculatiom();
        obj.sum(10,10,10);
        obj.sum(20,20);
    }
}

```

OUTPUT:

```

30
40

```

2) DIFFERENT TYPES OF ARGUMENTS :

```

class calculation
{
    void sum(int a,int b)
    {
        System.out.println(a+b);
    }

    void sum(double a,double b)
    {
        System.out.println(a+b);
    }

    public static void main(String args[])
    {
        calculation obj=new calculation();
        obj.sum(10,10);
        obj.sum(22.3,22.5);
    }
}

```

OUTPUT :

```

20
48.1

```

3) OVERLOADING MAIN() METHOD :

```

class overloading
{
    public static void main(int a)
    {
        System.out.println(a);
    }
    public static void main(String args[])
    {
        System.out.println("main method() inoked");
    }
}

```

```

        main(10);
    }
}

```

OUTPUT :

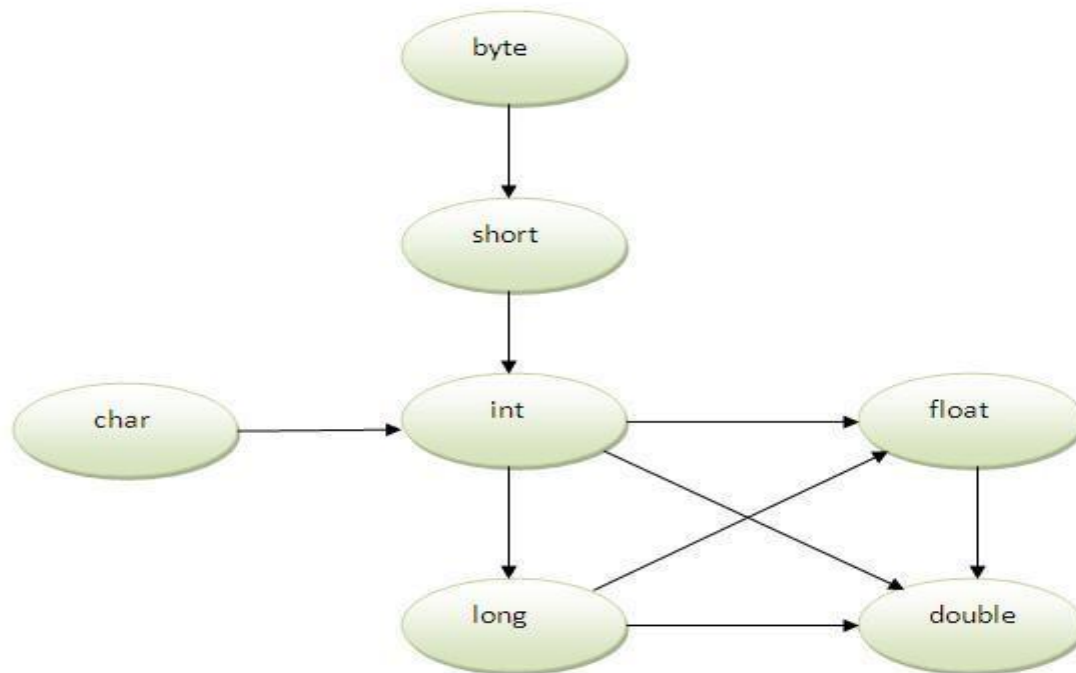
```

Main method() invoked
10

```

METHOD OVERLOADING AND TYPE PROMOTION

One type is promoted to another implicitly if no matching data type is found. Let's understand the concept by the figure given below:



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int, long, float or double and so on

```

class OverloadingCalculation1
{
    void sum(int a,long b)
    {
        System.out.println(a+b);
    }
    void sum(int a,int b,int c)
    {
        System.out.println(a+b+c);
    }

    public static void main(String args[])

```

```

    }
    OverloadingCalculation1 obj=new OverloadingCalculation1();
    obj.sum(20,20);//now second int literal will be promoted to long
    obj.sum(20,20,20);
}
}

```

OUTPUT:

40

60

2. EXPLAIN IN DETAIL ABOUT INHERITANCE IN JAVA

SYNOPSIS

INTRODUCTION
 USE OF INHERITANCE
 FORMS OF INHERITANCE
 WHY NO MULTIPLE INHERITANCE
 SUPER KEYWORD
 EXAMPLE PROGRAMS

INTRODUCTION:

The mechanism of deriving a new from a from an old one is called inheritance

The old class is known as the **base class** or **super class** or **parent class**

The new class is called the **subclass** or **derived class** or **child class**

The inheritance allows subclasses to inherit all the variables and methods of their parent classes

USE OF INHERITANCE:

Code Reusability

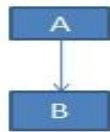
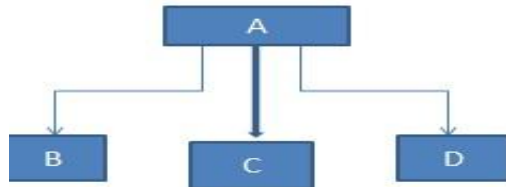
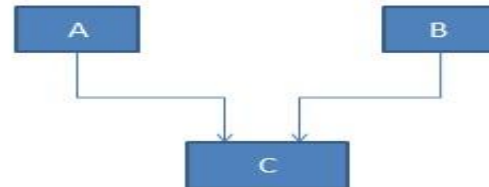
Method Overriding (so runtime polymorphism can be achieved).

FORMS OF INHERITANCE:

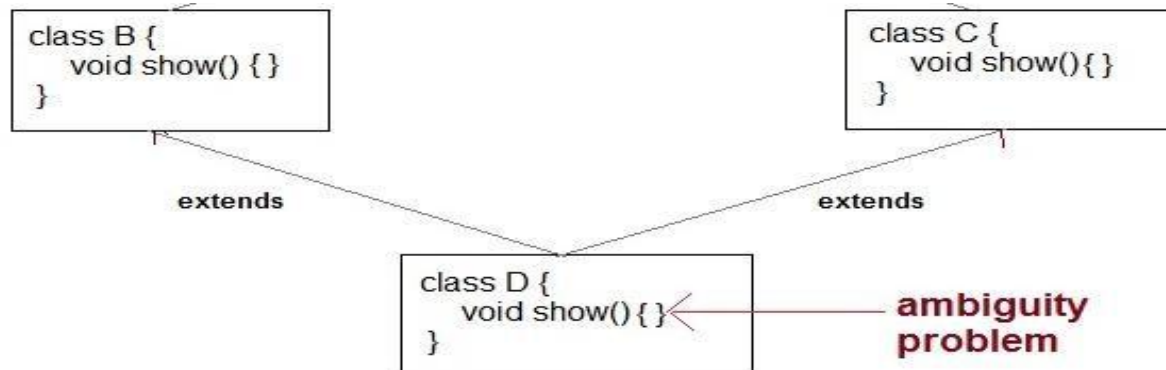
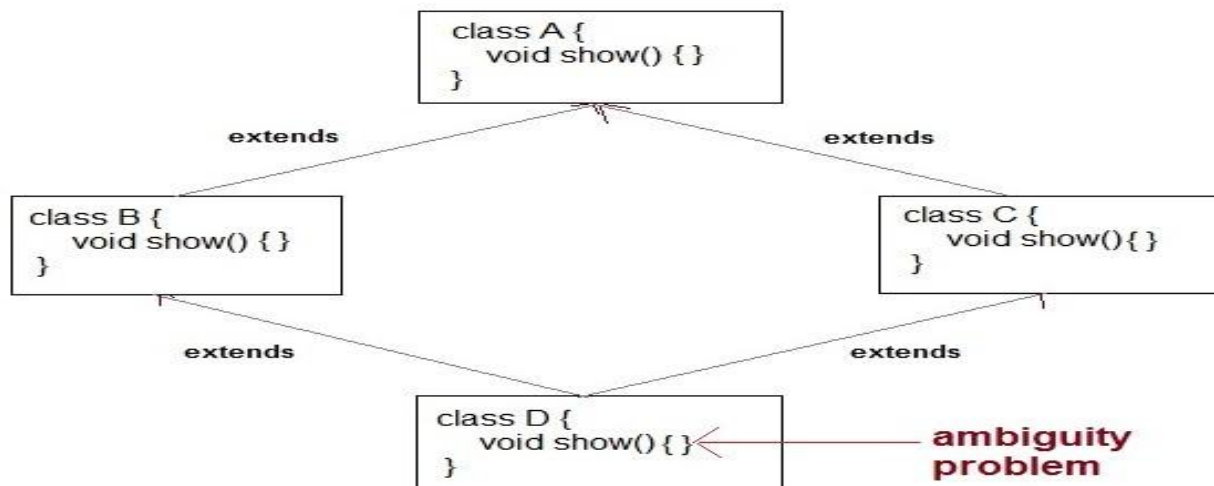
1. Single inheritance [only one super class]
2. Multiple inheritance [several super classes]
3. Hierarchical inheritance [one super classes, many subclasses]
4. Multilevel inheritance [derived from a derived class].

Java does not directly implement multiple inheritance but by using the concept of secondary inheritance path in the form of **interfaces**.

The diagrammatical representation of the forms of inheritance is as follows:

Single Inheritance:**Multilevel Inheritance****Hierarchical Inheritance:****Multiple Inheritance****WHY MULTIPLE INHERITANCE IS NOT SUPPORTED:**

- To reduce the complexity and simplify the language
- To remove ambiguity

TYPES OF AMBIGUITIES IN C++:**1) AMBIGUITY IN MULTIPLE INHERITANCE:****2) AMBIGUITY IN MULTI-PATH INHERITANCE:**

DEFINING A SUBCLASS :

- A Subclass can be defined as follows:

```
class subclassname extends superclassname
{
    Variable declaration;
    Methods declaration;
}
```

- The keyword **extends** signifies that the properties of the *superclassname* are extended to the *subclassname*.
- Now, the subclass contains its own variables and methods as well those of the superclass

EXAMPLE:

```
class Parent
{
    public void p1()
    {
        System.out.println("Parent method");
    }
}
public class Child extends Parent {
    public void c1()
    {
        System.out.println("Child method");
    }
    public static void main(String[] args)
    {
        Child cobj = new Child();
        cobj.c1();    //method of Child class
        cobj.p1();   //method of Parent class
    }
}
```

OUTPUT:

```
Parent Method
Child Method
```

SUPER KEYWORD:

- The **super** keyword in java is a reference variable that is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.

USE OF SUPER :

- super is used to refer immediate parent class instance variable
- super() is used to invoke immediate parent class constructor
- super is used to invoke immediate parent class method

1) **super is used to refer immediate parent class instance variable:**

```

class Vehicle
{
    int speed=50;
}
class Bike3 extends Vehicle
{
    int speed=100;
    void display()
    {
        System.out.println(super.speed); //will print speed of Vehicle
        System.out.println(speed); //will print speed of Bike
    }
    public static void main(String args[])
    {
        Bike3 b=new Bike3();
        b.display();
    }
}

```

OUTPUT:

```

50
100

```

2) **SUPER CAN BE USED TO INVOKE PARENT CLASS METHOD:**

```

class Person
{
    void message()
    {
        System.out.println("welcome");
    }
}

class Student extends Person
{
    void message()
    {
        System.out.println("welcome to java");
    }

    void display()
    {
        message(); //will invoke current class message() method
        super.message(); //will invoke parent class message() method
    }
}

```

```

public static void main(String args[])
{
    Student s=new Student();
    s.display();
}
}

```

OUTPUT:

```

Welcome to java
Welcome

```

3) SUPER IS USED TO INVOKE PARENT CLASS CONSTRUCTOR

```

class Vehicle
{
    Vehicle()
    {
        System.out.println("Vehicle is created");
    }
}

class Bike extends Vehicle
{
    Bike()
    {
        super();//will invoke parent class constructor
        System.out.println("Bike is created");
    }
    public static void main(String args[])
    {
        Bike b=new Bike();
    }
}

```

```

OUTPUT: Vehicle
is created Bike is
created

```

**3. EXPLAIN IN DETAIL ABOUT MULTITHREADING IN JAVA
MULTITHREADED PROGRAMMING**

Multithreading is a powerful programming tool that makes java distinctly different from its fellow programming languages.

- The ability of executing several programs simultaneously is called **Multitasking**.
- The ability of executing several processes simultaneously is called **Multithreading**.

THREAD

- **A Thread** is simple process
- A thread has a beginning, a body, and an end, and executes command sequentially.
- All the main program can be called single-threaded programs.
- Since threads in java share the same memory space, they are known as **Lightweight threads** or **Lightweight processes**.
- ‘Threads running in parallel’ does not mean they are running at the same time.

USES OF MULTITHREADING

- Enables programmers to do multi things at a time.
- A long program can be divided into threads and execute them in parallel.
- Extensively used in java-enabled browsers as Hot. Java

DIFFERENCE BETWEEN MULTITASKING AND MULTITHREADING

Sl. No	MULTITHREADING	MULTITASKING
1	It is a programming concept in which a program is divided into two or more threads executed in parallel	An operating system concept in which multiple tasks are performed simultaneously
2	It supports execution of multiple process simultaneously	It supports execution of multiple program simultaneously
3	The processor has to switch between different parts or threads of a program	The processor has to switch between different programs.
4	Highly efficient	Less efficient compared to multithreading.
5	A thread is the smallest unit in multithreading	A program is the smallest unit in multitasking
6	Helps in developing efficient programs	Helps in developing efficient OS.
7	It is cost-effective in case of context switching	Expensive in case of context switching.

CREATING THREADS

- Threads are implemented in the form of objects that contain a method called **run ()**.
- The run () method is the **heart and soul** of any thread.
- It makes up the entire body of the method.
- It's the only method in which the thread's behavior can be implemented.
- The run () method should be invoked by an object of the concerned thread.
- This can be done by creating a thread and initializing it with the help of another method called **start ()**.

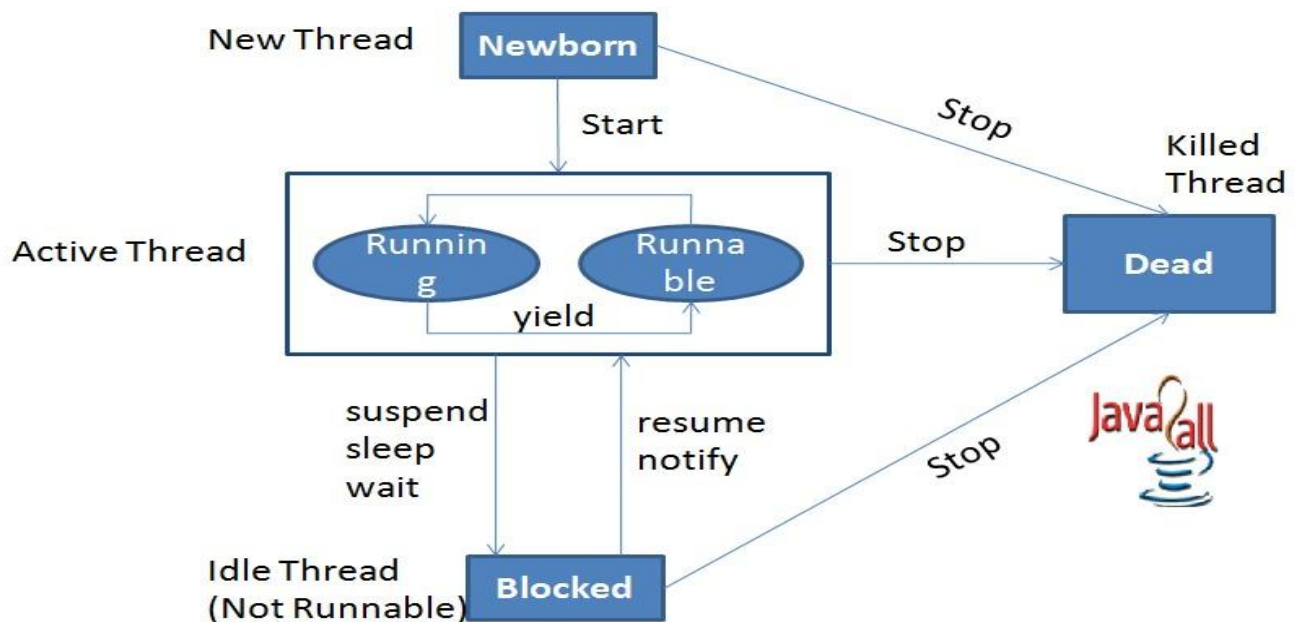
Syntax:

```
public void run ( )
{
..... (statements)
.....
}
```

LIFE CYCLE OF A THREAD

During the lifetime of a thread, it can enter many states as,

1. Newborn state.
2. Runnable state
3. Running state
4. Blocked state
5. Dead state



State transition diagram of a thread

NEWBORN STATE:

- When we create the Thread object, the thread is born and is said to be in **Newborn state**.
- At this state we can do only one of the following,
 1. Schedule it for running using **start ()** method.
 2. Kill it using **stop ()** method.

RUNNABLE STATE:

- The **runnable** state means that the thread is ready for execution and is waiting for the availability of the processor.
- The process of assigning time to threads is known as **time-slicing**.
- The first-come, first-serve manner is used here.

- If we want to relinquish control to another thread to equal the priority before it comes, we can use the **yield ()** method

RUNNING STATE :

- Running means that the processor has given its time to the thread for its execution.
- A running thread may relinquish its control in one of the following situations:
 1. It has been suspended using **suspend ()** method. A suspended thread can be resumed by using the **resume ()** method.
 2. It has been made to sleep by using the **sleep ()** method.
 3. It has been told to wait until some event occurs using the **wait ()** method .

BLOCKED STATE:

- A thread is said to be blocked when it is prevented from entering into the runnable state and the running state.
- A blocked thread is considered “ not runnable” but not dead, and is fully qualified to run again.

DEAD STATE:

- A running thread ends its life when it has completed executing its run () method.
- It is **natural death !**.
- A thread can be killed as soon as it is born, or while running or even when it is in “not runnable “ condition.

CREATING A NEW THREAD:

There are two ways,

1. **By creating a thread class** - Define a class that extends Thread class and override its **run ()** method.
2. **By converting a class to a thread** – Define a class that implements **Runnable** interface.

EXTENDING THE THREAD CLASS:

- We can extend the class by using **java.lang.Thread**.
- This gives access to all thread methods directly.
- It includes the following:
 1. Declare the class as extending the **Thread** class.
 2. Implement the **run ()** method for executing the Thread.
 3. Create a thread object and call the **start ()** method to initiate thread execution.

DECLARING THE CLASS :

The Thread class can be extended as follows:

```
class MyThread extends Thread
{
.....
.....
}
```

Now we have a new type of thread **MyThread**.

STARTING A NEW THREAD:

The syntax for creating and running an instance of the thread is as follows,

```
MyThread t1 = new MyThread();
t1.start();
```

First line - instantiates a new object of class MyThread.

Second Line - calls the **start ()** method causing the thread to move into the **runnable** state.

EXAMPLE PROGRAM:

```
import java.lang.*;

class MyThreadA extends Thread
{
    public void run()
    {
        for(int i=1;i<=10;i++)
        {
            System.out.println("MyThread 1 :"+i);
        }
    }
}

class MyThreadB extends Thread
{
    public void run()
    {
        for(int i=1;i<=5;i++)
        {
            System.out.println("MyThread 2 :"+i);
        }
    }
}

public class ThreadExample
{
    public static void main(String args[]) throws Exception
    {
        MyThreadA m1=new MyThreadA();
        MyThreadB m2=new MyThreadB();

        m1.start();
        m2.start();
    }
}
```

4. EXPLAIN IN DETAIL ABOUT FILES AND STREAM IN JAVA

INTRODUCTION:

- Java I/O (Input and Output) is used to process the input and produce the output based on the input.
- Java uses the concept of stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.
- We can perform file handling in java by java IO API.

STREAM

- A stream is a sequence of data. In Java a stream is composed of bytes. It's called a stream because it's like a stream of water that continues to flow.
- In java, 3 streams are created for us automatically. All these streams are attached with console
 - 1) System.out: standard output stream
 - 2) System.in: standard input stream
 - 3) System.err: standard error stream

Let's see the code to print output and error message to the console.

```
System.out.println("simple message");
System.err.println("error message");
```

Let's see the code to get input from console.

```
int i=System.in.read();//returns ASCII code of 1st character
System.out.println((char)i);//will print the character
```

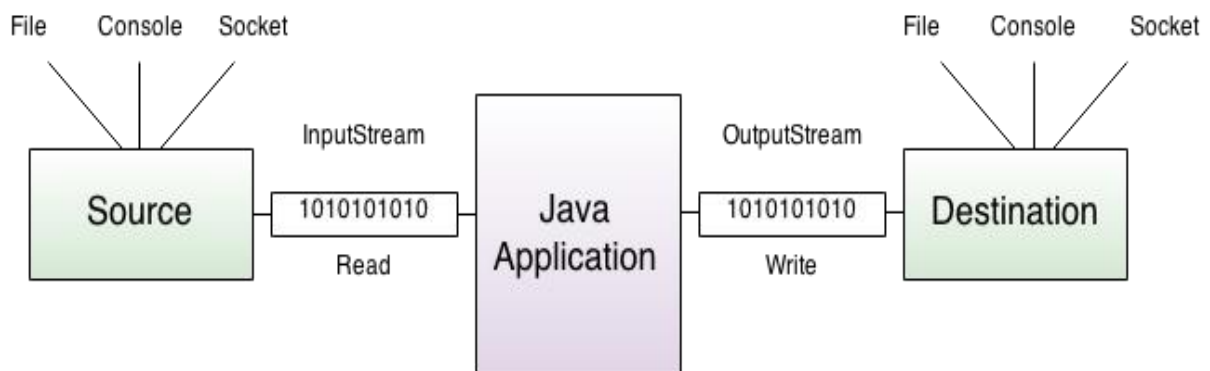
TYPES OF STREAM

OutputStream

Java application uses an output stream to write data to a destination, it may be a file, an array, peripheral device or socket.

InputStream

Java application uses an input stream to read data from a source, it may be a file, an array, peripheral device or socket.



STREAM CLASS**OutputStream class**

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

COMMONLY USED METHODS OF OUTPUTSTREAM CLASS

METHOD	DESCRIPTION
1) public void write(int) throws IOException:	is used to write a byte to the current output stream.
2) public void write(byte[]) throws IOException:	is used to write an array of byte to the current output stream.
3) public void flush() throws IOException:	flushes the current output stream.
4) public void close() throws IOException:	is used to close the current output stream.

InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Commonly used methods of OutputStream class

METHOD	DESCRIPTION
1) public abstract int read() throws IOException:	reads the next byte of data from the input stream. It returns -1 at the end of file.
2) public int available() throws IOException:	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close() throws IOException:	is used to close the current input stream.

FILE HANDLING STREAM CLASS

- FileOutputStream
- FileInputStream
- FileReader
- FileWriter

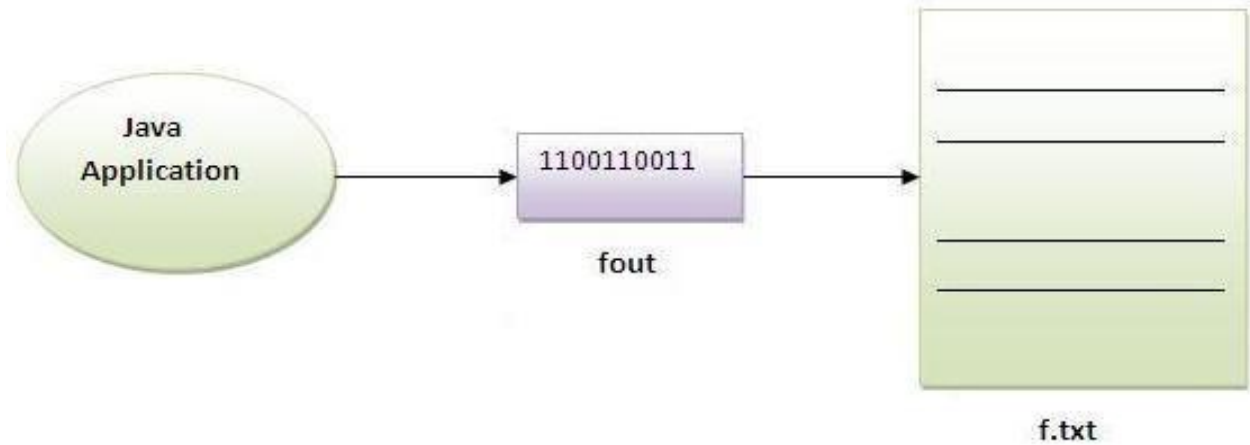
Prefer FileInputStream and FileOutputStream classes to read and write primitive values or byte-oriented data (for example to read image, audio, video etc.,) in a file

Prefer FileReader and FileWriter classes to read and write character-oriented data or text in a file

FileOutputStream can write byte-oriented as well as character - oriented data

FILEOUTPUTSTREAM

Java FileOutputStream is an output stream for writing data to a file

**FILEOUTPUTSTREAM****EXAMPLE:**

```

import java.io.*;
public class FDemo
{
    public static void main(String arg[]) throws IOException
    {
        FileOutputStream out=new FileOutputStream("Sample.txt");
        out.write(20);//writing integer value 20 in file
        //out.write("HI");//Not possible to write string with FOS object
        String s="HI";
        byte b[]=s.getBytes();//converting string to bytes
        out.write(b);
    }
}
  
```

USE OF PRINTSTREAM

```

import java.io.*;
public class FDemo
{
    public static void main(String arg[]) throws IOException
    {
        FileOutputStream out=new FileOutputStream("Sample.txt");
        PrintStream ps=new PrintStream(out);
        ps.println("HI");//using ps object writing HI in file
    }
}
  
```

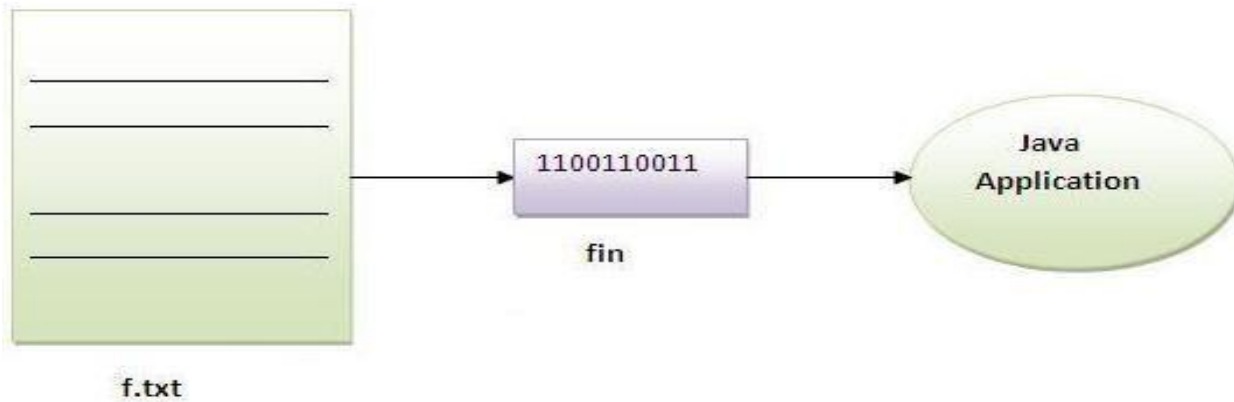
```

        ps.println("Hello");
    }
}

```

FILEINPUTSTREAM

- Java FileInputStream class obtains input bytes from a file.
- It is used for reading streams of raw bytes such as image data.
- For reading streams of characters, consider using FileReader



```

import java.io.*;
public class FDemo
{
    public static void main(String arg[]) throws IOException
    {
        FileInputStream in=new FileInputStream("Sample.txt");
        int a=in.read();//read() reads a character from a file
        System.out.println((char)a);
    }
}

```

EXAMPLE:

```

import
java.io.*; public
class FDemo
{
    public static void main(String arg[]) throws IOException
    {
        FileInputStream in=new FileInputStream("Sample.txt");
        int i;
        while((i=in.read())!=-1)//read() will return -1 if it reaches end of file
        {
            System.out.println((char)i);
        }
    }
}

```

}

USE OF BUFFEREDREADER**EXAMPLE:**

```

import java.io.*;
public class FDemo
{
    public static void main(String arg[]) throws IOException
    {
        FileReader in=new FileReader("Sample.txt");
        BufferedReader br=new BufferedReader(in);
        String s=br.readLine();//reads a line from the file
        System.out.println(s);
    }
}

```

5. EXPLAIN IN DETAIL ABOUT EXCEPTION HANDLING IN JAVA**SYNOPSIS**

EXCEPTION HANDLING IN JAVA
 EXCEPTION HANDLING
 ADVANTAGE OF EXCEPTION HANDLING
 HIERARCHY OF EXCEPTION CLASSES
 TYPES OF EXCEPTION
 SCENARIOS WHERE EXCEPTION MAY OCCUR
 KEYWORDS IN EXCEPTION HANDLING

The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

What is exception

Dictionary Meaning: Exception is an abnormal condition.

In java, exception is an event that disrupts the normal flow of the program.

WHAT IS EXCEPTION HANDLING

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IO, SQL, Remote etc.

ADVANTAGE OF EXCEPTION HANDLING

The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling.

Let's take a scenario:

1. statement 1;

2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the exception will be executed. That is why we use exception handling in java.

Types of Exception

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between checked and unchecked exceptions

1) Checked Exception

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that extend RuntimeException are known as unchecked exceptions e.g.ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Common scenarios where exceptions may occur

There are given some scenarios where unchecked exceptions can occur.

They are as follows:

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

2) Scenario where NullPointerException occurs

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

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

3) Scenario where `NumberFormatException` occurs

The wrong formatting of any value, may occur `NumberFormatException`. Suppose I have a string variable that have characters, converting this variable into digit will occur `NumberFormatException`.

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

4) Scenario where `ArrayIndexOutOfBoundsException` occurs

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

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

JAVA EXCEPTION HANDLING KEYWORDS

There are 5 keywords used in java exception handling.

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

JAVA TRY BLOCK

Java try block is used to enclose the code that might throw an exception. It must be used within the method.

Java try block must be followed by either catch or finally block

Syntax of java try-catch

```
try
{
    //code that may throw exception
}

catch(Exception_type obj)
{
}
}
```

JAVA CATCH BLOCK

Java catch block is used to handle the Exception. It must be used after the try block only.

You can use multiple catch block with a single try

EXAMPLE PROGRAM

```

public class Testtrycatch2
{
    public static void main(String args[])
    {
        try
        {
            int data=50/0;
        }
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code...");
    }
}

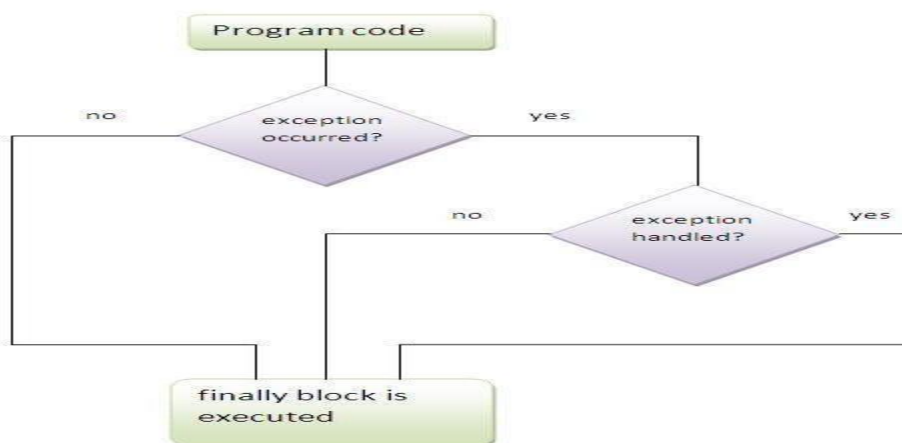
```

JAVA FINALLY BLOCK

Java finally block is a block that is used to *execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block must be followed by try or catch block

**JAVA THROW KEYWORD**

- The Java throw keyword is used to explicitly throw an exception.
- We can throw either checked or unchecked exception in java by throw keyword.
- The throw keyword is mainly used to throw custom exception

EXAMPLE PROGRAM

```

class ExceptionThrowDemo
{
    public static void main(String args[ ])
    {

```

```

        int a=Integer.parseInt(args[0]);
        try
        {
            if(a>10)
            {
                throw new ArithmeticException("A is greater than 10");
            }
        }
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        System.out.println("Program End");
    }
}

```

JAVA THROWS

- The **Java throws** keyword is used to declare an exception.
- It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained
- Exception Handling is mainly used to handle the checked exceptions
- If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used

Syntax:

```

return_type method_name() throws exception_class_name
{
    ...
}

```

UNIT 3 - 2 MARKS**1. DEFINE APPLICATION PROGRAMS AND APPLLET PROGRAMS:**

Using java we can develop two types of programs.

They are:

Application programs

Applet programs

Application programs are programs to do jobs on a local computer.

Applet programs are programs that have an ability to run on internet through a web browser.

2. DIFFERENCE BETWEEN APPLICATION AND APPLLET

APPLICATION	APPLET
1) Application can access the local file system and resources.	1) Restricted to access the local file system and resources.
2) Functionality of the applications are known.	2) Functionality of applets are not known.
3) Author is known.	3) Author is not known.
4) Creating and running an application is easy.	4) Creating and running an applet is complex.
5) This is executed by typing commands on command line.	5) This is executed by using applet viewer or any browser.

3. DEFINE APPLLET

Applets are small applications that are accessed on an internet server.

They are small, dynamic and graphical user interactive program that can execute inside a webpage displayed by JAVA capable browser such as IE, Netscape navigator etc

4. APPLLET TAG IN HTML

<applet> tag is used to start applet from inside the HTML document as well from the applet viewer.

Each **<applet>** tag may be executed in separate window by applet viewer.

BUT, the JAVA Capable browser can execute numerous applets inside a single webpage.

5. APPLLET LIFE CYCLE

There are four stages in the applet life cycle. They are

Born (or) Initialization State

Running state

Idle State

Dead/Destroyed State

6. WHAT ARE THE TYPES OF CONTAINERS IN JAVA AND MENTION THEM

There are two types of **AWT Container Classes**:

Top-Level Containers: Frame, Dialog and Applet

Secondary Containers: Panel and ScrollPane

7. MENTION ANY TWO FEATURES OF SWING

Swing is written in pure Java (except a few classes) and therefore is 100% portable.

Swing components are lightweight. The AWT components are heavyweight (in terms of system resource utilization).

8. WHAT IS GRAPHICS CLASS?

One of the Most important features of JAVA is its ability to draw GRAPHICS.

Every applet has its own area of the screen known as CANVAS (display area)

9. DEFINE AWT PACKAGES?

AWT is huge. It consists of 12 packages. But only 2 packages - java.awt and java.awt.event - are commonly-used.

1. The java.awt package contains the core AWT graphics classes:

- GUI Component classes (such as Button, TextField, and Label),
- GUI Container classes (such as Frame, Panel, Dialog and ScrollPane),
- Layout managers (such as FlowLayout, BorderLayout and GridLayout),
- Custom graphics classes (such as Graphics, Color and Font).

2. The java.awt.event package supports event handling:

- Event classes (such as ActionEvent, MouseEvent, KeyEvent and WindowEvent),
- Event Listener Interfaces (such as ActionListener, MouseListener, KeyListener and WindowListener),
- Event Listener Adapter classes (such as MouseAdapter, KeyAdapter, and WindowAdapter).

10. DIFFERENCE BETWEEN SWING AND AWT?

No.	Java AWT	Java Swing
1)	AWT components are platform-dependent .	Java swing components are platform-independent .
2)	AWT components are heavyweight .	Swing components are lightweight .
3)	AWT doesn't support pluggable look and feel .	Swing supports pluggable look and feel .
4)	AWT provides less components than Swing.	Swing provides more powerful components such as tables, lists, scrollpanes, colorchooser, tabbedPane etc.

11. NAME SOME COMMON METHODS USED IN AWT?

Method	Description
public void add(Component c)	inserts a component on this component.
public void setSize(int width,int height)	sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.

12.WHAT ARE THE GUI COMPONENTS AVAILABLE IN JAVA

Buttons

Adding buttons to a Frame or Panel

Action Listeners for Buttons

Inner Classes

Other GUI components

Labels

Text Fields

Text Areas

Assignment

13.WHAT IS SWING IN JAVA?

Swing is a set of program components for Java programmers that provide the ability to create graphical user interface (GUI) components, such as buttons and scroll bars, that are independent of the windowing system for specific operating system .

Swing components are used with the Java Foundation Classes (JFC).

Swing provides the look and feel of the modern java GUI

14.WRITE DOWN THE FEATURES OF SWING IN JAVA.

Swing Components are light weight

Swing supports a pluggable look and full

15.SPECIFY FEW COMPONENTS OF SWING

JEditorPanels and JPasswordFields - for displaying web pages and inputting confidential information

Dialogs - for displaying warnings, errors, prompting for input

Check Boxes - for selecting a small set of options

Radio Buttons - for mutually exclusive selection of options

Combo Boxes - for selecting a small set of qualified options

JLists - for selecting from a large set of qualified options

JTables - for table displays

JMenuBar, JMenu, and JMenuItem - for creating menus

File Chooser - for navigating through files and directories, and selecting files and directories

Tabbed Panels - for allowing different panels to occupy the same screen area.

16.WHAT IS JCOMPONENT

The class **JComponent** is the base class for all Swing components except top-level containers.

To use a component that inherits from JComponent, you must place the component in a containment hierarchy whose root is a top-level Swing container.

17.WHAT IS CONTAINER?

Container, in the context of Java development, refers to a part of the server that is responsible for managing the lifecycle of Web applications.

The Web applications specify the required lifecycle management with the help of a contract presented in XML format.

The Web container cannot be accessed directly by a client. Rather, the server manages the Web container, which in turn manages the Web application code.

18. WHAT ARE THE TYPES OF CONTAINER?**Two types:**

Top level container
Light weight container

19. MENTION SOME OF THE TOP LEVEL CONTAINER

JFrame, JDialog, JWindow and JApplet

20. MENTION SOME OF THE LIGHT WEIGHT CONTAINER

General purpose container. Eg: JPanel

A light weight container can be contained in another light weight container.

21. WHAT IS THE MAIN PACKAGE NEEDED TO RUN SWING IN JAVA?

Javax.swing

22. WHAT IS ADAPTER CLASS?

An adapter class provides the default implementation of all methods in an event listener interface.

Adapter classes are very useful when you want to process only few of the events that are handled by a particular event listener interface.

23. DEFINE EVENT HANDLING IN JAVA

When an application or a program keeps on monitoring and quickly responds to any action that occurs at the GUI interface, like mouse movement, selecting an item in a list or entering a keyboard input and so on then such a scenario is termed as event handling.

In java the events from the event sources are captured and they are sent to event listeners for respective actions to be taken.

24. WHAT IS DELEGATION EVENT MODEL.

The event handling mechanism used by swing is the same that used by AWT. This is called Delegation Event Model.

Event Delegation Model is based on four concepts:

The Event Classes
The Event Listeners
Explicit Event Enabling
Adapters

25. WHICH PACKAGE IS NEEDED TO IMPLEMENT EVENT HANDLING IN JAVA

Java.awt.event package is needed to implement event handling in java

26. SPECIFY SOME OF THE MOUSE RELATED EVENTS

Some of the mouse related events are

MouseEvent
MouseEvent
MouseEvent
MouseEvent

27. WHAT IS THE PURPOSE OF LAYOUT MANAGER?

Layouts are invoked using layout manager interface and set by the setLayout method.

Java LayoutManagers are used to save you the effort of manually putting Components where you want them on a Container.

Even more useful is the fact that a LayoutManager object will handle repositioning your objects whenever the Container is resized.

28. MENTION SOME OF THE TYPES OF LAYOUTS

Flow Layout

Border Layout

GridLayout

CardLayout

Gridbag Layout

29. DEFINE FLOW LAYOUT

FlowLayout is the default manager. It implements a simple layout style, which is similar to how words flow in a text editor.

Components are laid out from the upper left corner, left to right and top to bottom.

When no more components fit on a line, the next one appears on the next line.

A small space inleft between each component,above and below, as well as left and right.

The Constructors are:

FlowLayout()

FlowLayout(int how)

FlowLayout(int how,int horz,int vert)

30. DEFINE BORDER LAYOUT

The Border layout class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center.

The four sides are referred as north,south,east,west. The middle area is called the center.

The Constructors are,

BorderLayout() - creates a default border layout.

BorderLayout(int horz, int vert) - allows to specify the horizontal and vertical space between components in horz and vertical.

The constants, used to specify the regions:

BorderLayout.CENTER

BorderLayout.NORTH

BorderLayout.EAST

BorderLayout.WEST

BorderLayout.SOUTH

31. DEFINE GRID LAYOUT

Grid layout is used to lay out components in a two dimensional grid. We can define the number of rows and columns.

The constructors are,

GridLayout() - creates a single column grid layout.

GridLayout(int rows,int columns) - creates a grid layout with the specified number of rows and columns.

GridLayout(int rows,int columns,int horz,int vert) – creates a grid layout with the specified number of rows and columns and allows to specify the horizontal and vertical space between components in horz and vertical respectively.

32.DEFINE CARD LAYOUT

Card layout is unique from other layout manager useful for user interfaces

33.WRITE ABOUT GRAPHICS IN JAVA

Graphics object encapsulates state information needed for the basic rendering operations that Java supports.

This state information includes the following properties:

The Component object on which to draw

A translation origin for rendering and clipping coordinates.

The current clip.

The current color.

The current font.

The current logical pixel operation function (XOR or Paint).

The current XOR alternation color (see setXORMode(java.awt.Color)).

34.WHICH CLASS AND WHAT ARE THE WAYS TO OBTAIN GRAPHICS IN JAVA

A graphics context is encapsulated by the graphics class and obtained in two ways

It is passed to an applet when one of its various methods such as paint() or update() is called

It is returned by the getGraphics() method of component.

35.METHODS OF A GRAPHICS CLASS

`g.drawLine(x1 , y1 , x2 , y2);`

`g.drawOval(left , top , width , height);`

`g.fillOval(left , top , width , height);`

`g.drawRoundRect(left , top , width , height);`

`g.fillRoundRect(left , top , width , height);`

`g.drawArc(left , top , width , height , startAngle , arcAngle);`

`g.drawString(string , x , y);`

36.What is source and listener?

source: A source is an object that generates an event. This occurs when the internal state of that object changes in some way.

listener : A listener is an object that is notified when an event occurs. It has two major requirements.

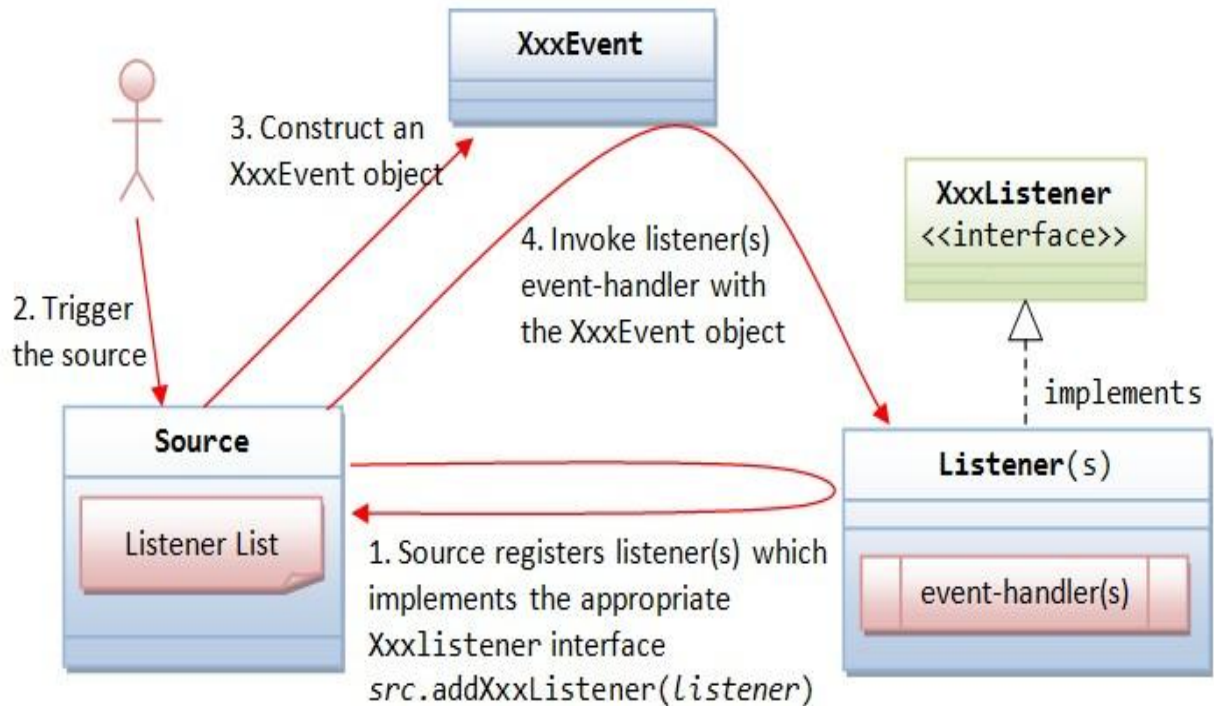
First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.

37.STATE THE STEPS TO PERFORM EVENT HANDLING

Following steps are required to perform event handling:

Implement the Listener interface and overrides its methods

Register the component with the Listener

38. ILLUSTRATE THE WORKING OF EVENT HANDLING**38. WHAT IS MEANT BY CONTROLS AND WHAT ARE DIFFERENT TYPES OF CONTROLS IN AWT?**

Controls are components that allow a user to interact with your application and the AWT supports the following types of controls:

Labels, Push Buttons, Check Boxes, Choice Lists, Lists, Scrollbars, Text Components.

39. WHAT IS THE DIFFERENCE BETWEEN SCROLLBAR AND SCROLLPANE?

A Scrollbar is a Component, but not a Container whereas Scrollpane is a Container and handles its own events and perform its own scrolling.

40. WHAT IS A LAYOUT MANAGER AND WHAT ARE DIFFERENT TYPES OF LAYOUT MANAGERS AVAILABLE IN JAVA.AWT?

A layout manager is an object that is used to organize components in a container. The different layouts are available are FlowLayout, BorderLayout, CardLayout, GridLayout and GridBagLayout.

41. HOW ARE THE ELEMENTS OF DIFFERENT LAYOUTS ORGANIZED?

FlowLayout: The elements of a FlowLayout are organized in a top to bottom, left to right fashion.

BorderLayout: The elements of a BorderLayout are organized at the borders (North, South, East and West) and the center of a container.

CardLayout: The elements of a CardLayout are stacked, on top of the other, like a deck of cards.

GridLayout: The elements of a GridLayout are of equal size and are laid out using the square of a grid.

GridBagLayout: The elements of a GridBagLayout are organized according to a grid. However, the elements are of different size and may occupy more than one row or column of the grid. In addition, the rows and columns may have different sizes.

42. WHICH CONTAINERS USE A BORDER LAYOUT AS THEIR DEFAULT LAYOUT?

Window, Frame and Dialog classes use a BorderLayout as their layout.

43. WHICH CONTAINERS USE A FLOW LAYOUT AS THEIR DEFAULT LAYOUT?

Panel and Applet classes use the FlowLayout as their default layout.

39. WHAT IS EVENT HANDLER?

When an application or a program keeps on monitoring and quickly responds to any action that occurs at the GUI interface, like mouse movement, selecting an item in a list or entering a keyboard input and so on then such a scenario is termed as event handling.

In JAVA the events from the event sources are captured and they are sent to event listeners for respective actions to be taken.

40. WHAT IS AN APPLLET?

Applets are JAVA programs that may be embedded into HTML documents. The applet runs in the page as soon as it has been downloaded.

More specifically JAVA applets are JAVA programs that inherit the Applet (awt) or JApplet class (swing). There are two types of applet.

- Local applet
- Remote applet

41. WHAT IS GARBAGE COLLECTION? WHAT IS THE PROCESS THAT IS RESPONSIBLE FOR DOING THAT IN JAVA?

Reclaiming the unused memory by the invalid objects. Garbage collector is responsible for this process

42. WHAT KIND OF THREAD IS THE GARBAGE COLLECTOR THREAD? - It is a daemon thread**43. HOW CAN YOU FORCE GARBAGE COLLECTION?**

You can't force GC, but could request it by calling System.gc(). JVM does not guarantee that GC will be started immediately

44. HOW CAN YOU MINIMIZE THE NEED OF GARBAGE COLLECTION AND MAKE THE MEMORY USE MORE EFFECTIVE?

Use object pooling and weak object references

45. CAN YOU WRITE A JAVA CLASS THAT COULD BE USED BOTH AS AN APPLLET AS WELL AS AN APPLICATION?

Yes. Add a main() method to the applet

46. LIST THE AWT CONTROLS?

- Label
- Button
- Checkbox
- TextComponent
- Choice
- List
- Scrollbar

47. WRITE A NOTE ON PUSH BUTTON CONTROL?

A push button is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type Button. Button defines these two constructors.

```
Button()  
Button( String str)
```

The first version creates an empty button. The second creates a button that contains str as a label.

48. WRITE A NOTE ON BORDERLAYOUT?

The BorderLayout class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north, south, east, and west. The middle area is called the center. Here are the constructors defined by BorderLayout

```
BorderLayout()  
BorderLayout(int horz, int vert)
```

49. WRITE A NOTE ON CHECK BOX CONTROL IN JAVA?

A check box is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. You change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group. Check boxes are objects of the Checkbox class.

```
Checkbox()  
Checkbox( String str)  
Checkbox( String str, Boolean on)  
Checkbox( String str, Boolean on, CheckboxGroup cbGroup)
```

50. DISTINGUISH BETWEEN COMPONENT AND CONTAINER

Component is an abstract class that encapsulates all of the attributes of a visual component. All user interface elements that are displayed on the screen and that interact with the user are subclasses of Component.

The container class is a subclass of Component. It has additional methods that allow other Component objects to be nested within it. Other Container objects can be stored inside of a container.

11 MARKS**1. EXPLAIN IN DETAIL ABOUT ABSTRACT WINDOWING TOOLKIT IN JAVA WITH EXAMPLE PROGRAM****SYNOPSIS**

INTRODUCTION

PROGRAMMING GUI WITH AWT

AWT PACKAGES

CONTAINERS AND COMPONENTS

AWT CONTAINER CLASSES

AWT COMPONENT CLASSES

EXAMPLE PROGRAM

INTRODUCTION

There are two sets of Java APIs for GUI programming:

AWT (Abstract Windowing Toolkit) and Swing.

AWT API was introduced in JDK 1.0. Most of the AWT components have become obsolete and should be replaced by newer Swing components.

AWT API consists of set of classes needed to create GUI application in java

Swing API, a much more comprehensive set of graphics libraries that enhances the AWT, was introduced as part of Java Foundation Classes (JFC) after the release of JDK 1.1.

JFC, which consists of Swing, Java2D, Accessibility API, Internationalization, and Pluggable Look-and-Feel Support, was an add-on to JDK 1.1 but has been integrated into core Java since JDK 1.2.

Programming GUI with AWT:

Java Graphics APIs - AWT and Swing - provide a huge set of reusable GUI components, such as button, text field, label, choice, panel and frame for building GUI applications. You can simply reuse these classes rather than re-invent the wheels.

Many AWT classes are now obsolete and they are used only in exceptional circumstances.

AWT Packages

AWT is huge. It consists of 12 packages. But only 2 packages - java.awt and java.awt.event - are commonly-used.

1. The java.awt package contains the core AWT graphics classes:
 - GUI Component classes (such as Button, TextField, and Label),
 - GUI Container classes (such as Frame, Panel, Dialog and ScrollPane),
 - Layout managers (such as FlowLayout, BorderLayout and GridLayout),
 - Custom graphics classes (such as Graphics, Color and Font).

2. The java.awt.event package supports event handling:

- Event classes (such as ActionEvent, MouseEvent, KeyEvent and WindowEvent),
- Event Listener Interfaces (such as ActionListener, MouseListener, KeyListener and WindowListener),
- Event Listener Adapter classes (such as MouseAdapter, KeyAdapter, and WindowAdapter).

AWT provides a platform-independent and device-independent interface to develop graphic programs that runs on all platforms, such as Windows, Mac, and Linux.

CONTAINERS AND COMPONENTS:

There are two types of GUI elements:

1. **Component:** Components are elementary GUI entities (such as Button, Label, and TextField.)
2. **Container:** Containers (such as Frame, Panel and Applet) are used to hold components in a specific layout. A container can also hold sub-containers.

In the below example, there are three containers: a Frame and two Panels.

A Frame is the top-level container of an AWT GUI program. A Frame has a title bar (containing an icon, a title, and the minimize/maximize(restore-down)/close buttons), an optional menu bar and the content display area.

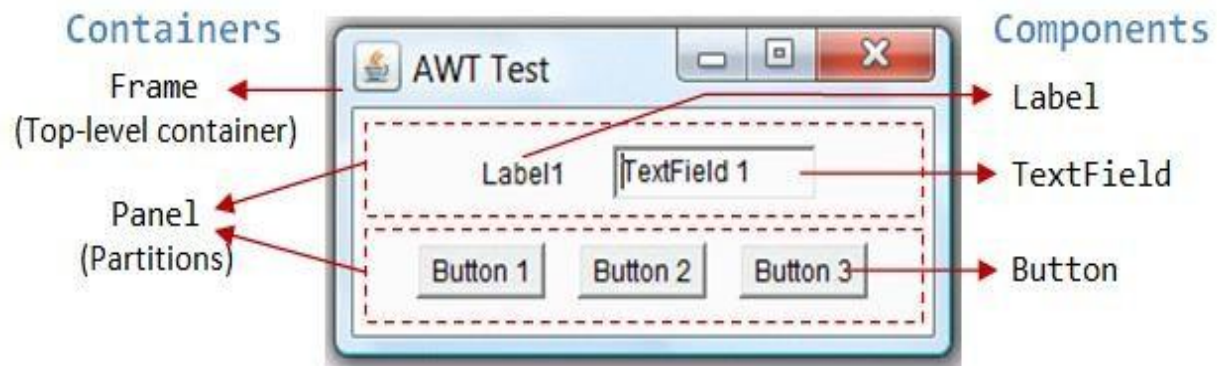
A Panel is a rectangular area (or partition) used to group related GUI components in a certain layout. In the above example, the top-level Frame contains two Panels.

There are five components: a Label (providing description), a TextField (for users to enter text), and three Buttons (for user to trigger certain programmed actions).

In a GUI program, a component must be kept in a container. You need to identify a container to hold the components. Every container has a method called add(Component c). A container (says aContainer) can invoke aContainer.add(aComponent) to add aComponent into itself.

For example,

```
Panel panel = new Panel();           // Panel is a Container
Button btn = new Button("Press");    // Button is a Component
panel.add(btn);                       // The Panel Container adds a Button Component
```



AWT CONTAINER CLASSES

There are two types of Containers. They are

- i) Top Level Containers
- ii) Secondary Level Containers

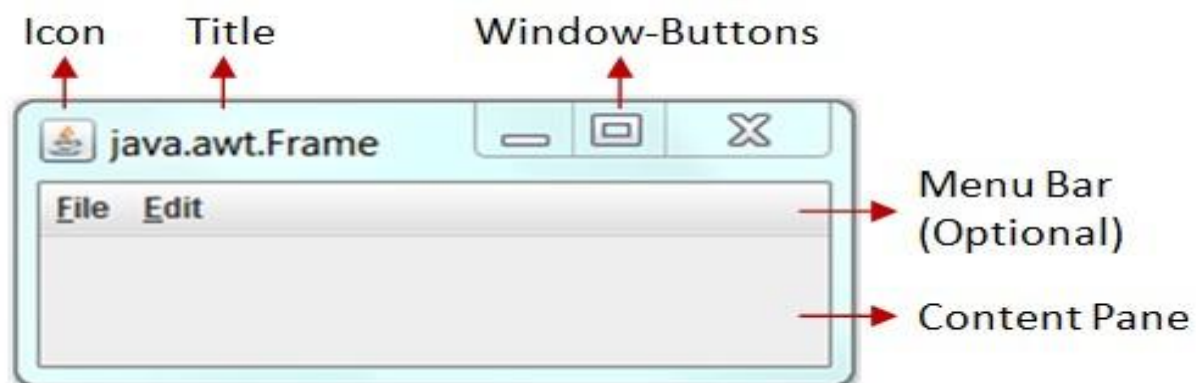
i) TOP-LEVEL CONTAINERS: Frame, Dialog and Applet

Each GUI program has a *top-level container*. The commonly-used top-level containers in AWT are Frame, Dialog and Applet:

Frame:

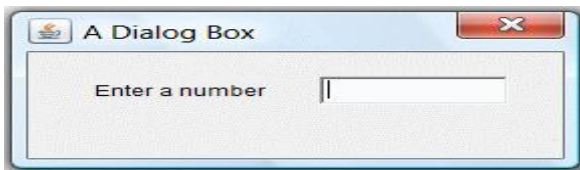
A Frame provides the "main window" for the GUI application, which has a title bar (containing an icon, a title, minimize, maximize/restore-down and close buttons), an optional menu bar, and the content display area.

To write a GUI program, we typically start with a subclass extending from `java.awt.Frame` to inherit the main window as follows:



Dialog:

An AWT Dialog is a "pop-up window" used for interacting with the users. A Dialog has a title-bar (containing an icon, a title and a close button) and a content display area, as illustrated.



Applet:

An AWT Applet (in package `java.applet`) is the top-level container for an applet, which is a Java program running inside a browser.

SECONDARY CONTAINERS: Panel and ScrollPane

Secondary containers are placed inside a top-level container or another secondary container.

AWT also provide these secondary containers:

Panel: a rectangular box (partition) under a higher-level container, used to *layout* a set of related GUI components. See the above examples for illustration.

ScrollPane: provides automatic horizontal and/or vertical scrolling for a single child component.

AWT COMPONENT CLASSES

AWT provides many ready-made and reusable GUI components. The frequently-used are: Button, TextField, Label, Checkbox, CheckboxGroup (radio buttons), List, and Choice, as illustrated below.



AWT Components:

Label

A `java.awt.Label` provides a text description message. Take note that `System.out.println()` prints to the system console, not to the graphics screen. You could use a Label to label another component (such as text field) or provide a text description.

Button

A `java.awt.Button` is a GUI component that triggers a certain programmed *action* upon clicking.

TextField

A `java.awt.TextField` is single-line text box for users to enter texts. (There is a multiple-line text box called `TextArea`.) Hitting the "ENTER" key on a `TextField` object triggers an action-event.

EXAMPLE PROGRAM:

```
import java.awt.*;

public class AWTDemo
{
    public static void main(String args[])
    {
        Frame f=new Frame("AWTDemo");
        f.setSize(500, 500);
        f.setVisible(true);

        Panel p=new Panel();

        Label l=new Label("VALUE A");
        Button b=new Button("ADD");
        TextField tf=new TextField(50);

        p.add(l);
        p.add(tf);
        p.add(b);

        f.add(p);
    }
}
```



2. EXPLAIN IN DETAIL ABOUT APPLET PROGRAMMING IN JAVA**SYNOPSIS**

DEFINITION

CATEGORIES OF APPLET

STEPS TO CREATE AND EXECUTE AN APPLET

APPLET TAG IN HTML

APPLET LIFE CYCLE

COLOR CONTROL

FONT CONTROL

GRAPHICS CLASS

EXAMPLE PROGRAM

DEFINITION

Applets are small applications that are accessed on an internet server.

They are small, dynamic and graphical user interactive program that can execute inside a webpage displayed by JAVA capable browser such as IE, Netscape navigator etc.

Application :

Animation
multimedia presentation
Real time games

CATEGORIES OF APPLET

- i) Local Applet
- ii) Remote Applet

Local Applet :

Developed locally and stored in a local system.
Internet connection is not necessary

Remote Applet :

Developed externally and store in a remote computer.
Internet connection is necessary.

STEPS TO CREATE AND EXECUTE AN APPLET

Typing and saving the source code

Eg : c:\jdk 1.2\bin>edit appl.java

Compile the source code (Program)

Eg : c:\jdk1.2\bin>javac appl.java
(now the class file (appl.class) is created)

Create a .html file and write html code for getting class file for appl.class

Eg : `c:\jdk 1.2\bin>edit appl.html`

Example code:

```
<HTML>
<applet code = "appl.class" height=400 width=400 >
</applet>
</html>
```

To execute using applet viewer :

Eg: `C:\jdk1.2\bin>appletviewer appl.html`

APPLET TAG IN HTML

<applet> tag is used to start applet from inside the HTML document as well from the applet viewer.

Each **<applet>** tag may be executed in separate window by applet viewer.

BUT, the JAVA Capable browser can execute numerous applets inside a single webpage.

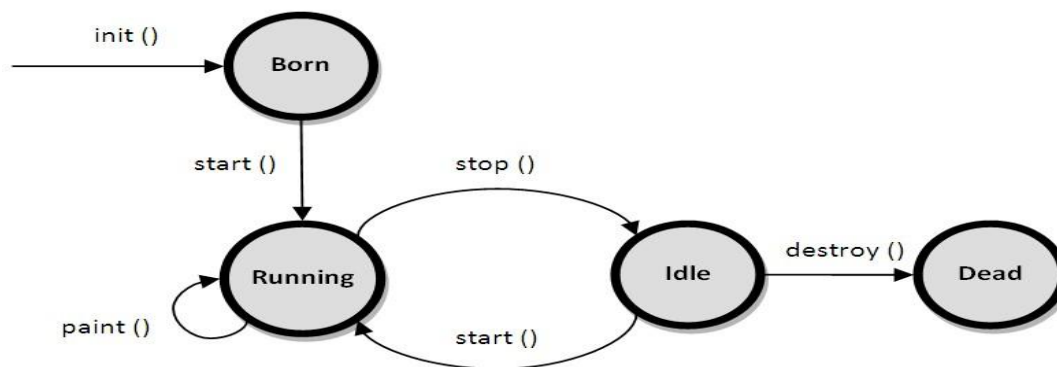
Attributes of <applet> tag:

Height
Width,
Code etc.,

APPLET LIFE CYCLE

There are four stages in the applet life cycle. They are

Born (or) Initialization State
Running state
Idle State
Dead/Destroyed State



BORN (OR) INITIALIZATION STATE

Whenever an applet is loaded it enters into the initialization state. This is achieved by calling the `init()` method .

It occurs only once in an applet lifecycle.

SYNTAX:

```
public void init()
{
.....
.....( Action)
.....
}
```

RUNNING STATE

Applet enters the running state when the system calls the **start()** method of the applet class.

The start() method is also executed automatically after the applet is initialised.

The start () method can be called several times.

SYNTAX :

```
public void start()
{
.....
..... (Action)
.....
}
```

DISPLAY STATE

An Applet moves to the display state whenever it has to perform some output operations on the screen.

The **paint()** method is used to accomplish this task when it enters the running state.

SYNTAX

```
public void paint(Graphics g)
{
.....
.....// Display Statements
}
```

IDLE (OR) STOPPED STATE

When we leave the page containing the current applet, it enters the idle/stopped state automatically.

We can also stop the applet explicitly by calling the **stop()** method.

While using a thread to run an applet , it is mandatory to use stop() method to terminate the thread.

SYNTAX

```
public void stop()
{
.....
..... ( Action)
}
```

DEAD (OR) DESTROYED STATE

An applet is said to be dead when it is removed from the memory.

This state also occurs automatically by invoking the `destroy()` method when we quit the browser.

It occurs only once like the initialization state.

SYNTAX

```
public void destroy()
{
.....
.....(Action)
}
```

COLOR CONTROL

Color is one of the classes from `java.awt.*` package.

Using this class, we can include more colors to the applet programs.

Example :

```
setBackgroundColor (Color.blue);
setForegroundColor (Color.white);
setColor(Color.green);
getColor();
```

FONT CONTROL

Different types of fonts are available in the font class in `java.awt.*`;

Some of the available fonts are :

Example :

```
Times New Roman
Verdana
Serif etc
```

GRAPHICS CLASS

One of the Most important features of JAVA is its ability to draw GRAPHICS.

Every applet has its own area of the screen known as CANVAS (display area)

Methods Of A Graphics Class

```

g.drawLine( x1 , y1 , x2 , y2 );
g.drawOval( left , top , width , height );
g.fillOval( left , top , width , height );
g.drawRoundRect( left , top , width , height );
g.fillRoundRect( left , top , width , height );
g.drawArc( left , top , width , height , startAngle , arcAngle );
g.drawString( string , x , y );

```

Example Program:

```

import java.awt.*;
import java.applet.*;
/*
<applet code="Shapes" width=800 height=250>
</applet>
*/

public class Shapes extends Applet
{
    public void init()
    {
        System.out.println("Initialize");
    }

    public void start()
    {
        System.out.println("Start");
    }

    public void paint(Graphics g)
    {
        g.drawString("SHAPES",200,20);
        g.setColor(Color.blue);
        g.drawLine(10,30,400,30);
        g.drawOval(20,50,150,100);
        g.fillRect(200,50,200,100);
    }

    public void stop()
    {
        System.out.println("Stop");
    }

    public void destroy()
    {
        System.out.println("Destroy");
    }
}

```

}

3. EXPLAIN IN DETAIL ABOUT THE SWING COMPONENTS IN JAVA.

SWING :

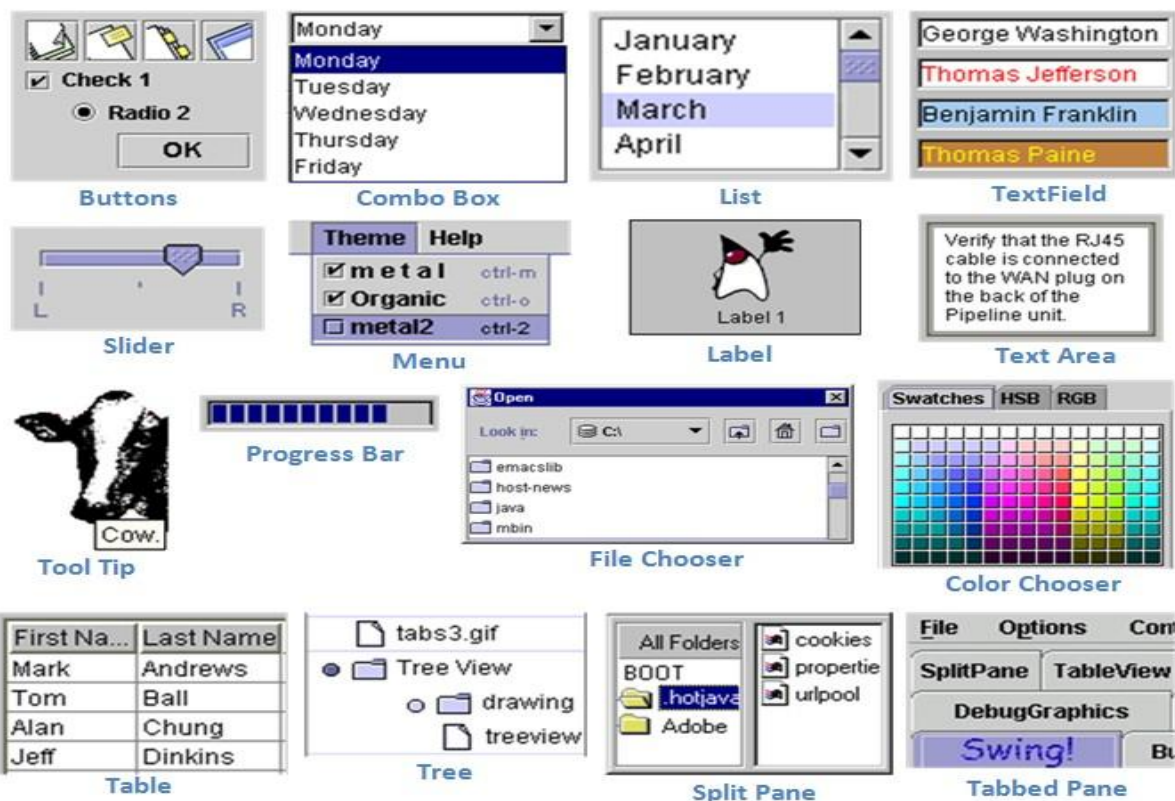
Java Swing is a GUI toolkit for Java. Swing is one part of the Java Foundation Classes (JFC). Swing includes graphical user interface (GUI) widgets such as text boxes, buttons, split-panes, and tables.

Swing widgets provide more sophisticated GUI components than the earlier Abstract Window Toolkit. Since they are written in pure Java, they run the same on all platforms, unlike the AWT which is tied to the underlying platform's windowing system.

Swing supports pluggable look and feel – not by using the native platform's facilities, but by roughly emulating (imitate) them. This means you can get any supported look and feel on any platform.

Swing's Features:

Swing is huge and has great depth. Compared with AWT, Swing provides a huge and comprehensive collection of reusable GUI components, as shown in the Figure below



Main Features:

Swing is written in pure Java (except a few classes) and therefore is 100% portable.

Swing components are lightweight. The AWT components are heavyweight (in terms of system resource utilization).

Each AWT component has its own opaque native display, and always displays on top of the lightweight components.

AWT components rely heavily on the underlying windowing subsystem of the native operating system.

For example, an AWT button ties to an actual button in the underlying native windowing subsystem, and relies on the native windowing subsystem for their rendering and processing.

Swing components (JComponents) are written in Java. They are generally not "weight-down" by complex GUI considerations imposed by the underlying windowing subsystem.

Swing components support pluggable look-and-feel. You can choose between Java look-and-feel and the look-and-feel of the underlying OS (e.g., Windows, UNIX or Mac).

If the later is chosen, a Swing button runs on the Windows looks like a Windows' button and feels like a Window's button. Similarly, a Swing button runs on the UNIX looks like a UNIX's button and feels like a UNIX's button.

Swing supports mouse-less operation, i.e., it can operate entirely using keyboard.

Swing components support "tool-tips".

SWING COMPONENTS:

Various JComponents available in SWING package:

JPanel

Jpanel is Swing's version of the AWT class Panel and uses the same default layout, FlowLayout. JPanel is descended directly from JComponent.

CONSTRUCTORS:

JPanel ()

Constructs a new blank JPanel

METHODS:

void paint (Graphics g)

Overrides the paint method of JPanel.

void repaint ()

Repaints this component, and causes a call to the paint method.

void setLayout (LayoutManager Manager)

Sets the layout manager for the panel.

Component add (Component Item)

Appends the specified component to the end of this container.

void add (Component Item, Object Constraints)

Appends the specified component with the specified constraints.

JFrame

JFrame is Swing's version of Frame and is descended directly from that class. The components added to the frame are referred to as its contents; these are managed by the contentPane. To add a component to a JFrame, we must use its contentPane instead.

CONSTRUCTORS:**JFrame ()**

Creates a blank Button instance.

JFrame (String Text)

Creates a Frame instance with the specified text in the title bar

METHODS:**void addWindowListener(WindowListener Handler)**

Configures a window event handler for the frame.

Container getContentPane()

Returns the contentPane object for this frame.

void setBackground (Color BackgroundColor)

Sets the background color of the frame.

void setFont (Font TextFont)

Sets the font for this component.

void setForeground (Color TextColor)

Sets the color of the text for the frame.

void setSize (int Width, int Height)

Resizes this window so that it has the specified Width and Height.

void setTitle (String Text)

Sets the text for the title bar.

void show ()

Makes the window visible.

JLabel

JLabel descended from JComponent, is used to create text labels.

CONSTRUCTORS:**JLabel ()**

Creates a blank JLabel instance.

JLabel (String Text)

Creates a JLabel instance with the specified text.

JLabel (String Text, int Alignment)

Creates a JLabel instance with the specified text and horizontal alignment

METHODS:**void setBackground (Color BgdColor)**

Sets the color of the background for the label.

void setHorizontalAlignment (int Alignment)

Sets the alignment of the label's contents along the X axis.

void setFont (Font TextFont)

Sets the font for this component. Arguments

void setForeground (Color TextColor)

Sets the color of the text for the label.

void setText (String Text)

Sets the text for the label.

JButton

JButton is a component the user clicks to trigger a specific action.

The abstract class **AbstractButton** extends class **JComponent** and provides a foundation for a family of button classes, including **JButton**

CONSTRUCTORS:**JButton ()**

Creates a blank **JButton** instance.

JButton (String Text)

Creates a **JButton** instance with the specified text.

METHODS:**void addActionListener (ActionListener Handler)**

Configures an event handler for the button.

void setActionCommand (String ActionText)

Sets the text for the action event of the button.

void setBackground (Color BackgroundColor)

Sets the background color of the button.

void setEnabled (boolean State)

Enables or disables the button.

void setFont (Font TextFont)

Sets the font for this component.

void setForeground (Color TextColor)

Sets the color of the text for the button.

void setText (String Text)

Sets the text for the button.

JTextField

JTextField allows editing of a single line of text. New features include the ability to justify the text left, right, or center, and to set the text's font.

CONSTRUCTORS:**JTextField ()**

Creates a blank **JTextField** instance.

JTextField (String Text)

Creates a **JTextField** instance with the specified text.

JTextField (int Columns)

Creates a blank JTextField instance with the specified number of columns.

JTextField (String Text, int Columns)

Creates a JTextField instance with the specified text and the specified number of columns.

METHODS:**void addActionListener (ActionListener Handler)**

Configures an event handler for the TextField.

String getText ()

Returns the text in the field.

void setBackground (Color BackgroundColor)

Sets the background color of the TextField.

void setEditable (boolean Editable)

Sets the field as being editable or fixed.

void setFont (Font TextFont)

Sets the font for this component.

void setText (String Text)

Sets the text for the field.

OTHER JCOMPONENT CLASSES:**JInternalFrame**

JInternalFrame is confined to a visible area of a container it is placed in. It can be iconified , maximized and layered.

JWindow

JWindow is Swing's version of Window and is descended directly from that class. Like Window, it uses BorderLayout by default.

JDialog

JDialog is Swing's version of Dialog and is descended directly from that class. Like Dialog, it uses BorderLayout by default. Like JFrame and JWindow,

JDialog contains a rootPane hierarchy including a contentPane, and it allows layered and glass panes. All dialogs are modal, which means the current thread is blocked until user interaction with it has been completed.

JDialog class is intended as the basis for creating custom dialogs; however, some of the most common dialogs are provided through static methods in the class JOptionPane.

JPasswordField

JPasswordField (a direct subclass of JTextField) you can suppress the display of input.

Each character entered can be replaced by an echo character.

This allows confidential input for passwords, for example. By default, the echo character is the asterisk, *.

JTextArea

JTextArea allows editing of multiple lines of text. **JTextArea** can be used in conjunction with class **JScrollPane** to achieve scrolling.

The underlying **JScrollPane** can be forced to always or never have either the vertical or horizontal scrollbar;

JRadioButton

JRadioButton is similar to **JCheckBox**, except for the default icon for each class. A set of radio buttons can be associated as a group in which only one button at a time can be selected.

JCheckBox

JCheckBox is not a member of a checkbox group. A checkbox can be selected and deselected, and it also displays its current state.

JComboBox

JComboBox is like a drop down box. You can click a drop-down arrow and select an option from a list.

For example,

When the component has focus, pressing a key that corresponds to the first character in some entry's name selects that entry. A vertical scrollbar is used for longer lists.

JList

JList provides a scrollable set of items from which one or more may be selected. **JList** can be populated from an Array or Vector.

JList does not support scrolling directly, instead, the list must be associated with a scrollpane.

The view port used by the scroll pane can also have a user-defined border. **JList** actions are handled using **ListSelectionListener**.

JTabbedPane

JTabbedPane contains a tab that can have a tool tip and a mnemonic, and it can display both text and an image.

JToolBar

JToolBar contains a number of components whose type is usually some kind of button which can also include separators to group related components within the toolbar.

JMenuBar

JMenuBar can contain several JMenu's. Each of the JMenu's can contain a series of JMenuItem 's that you can select. Swing provides support for pull-down and popup menus.

Scrollable JPopupMenu

Scrollable JPopupMenu is a scrollable popup menu that can be used whenever we have so many items in a popup menu that exceeds the screen visible height.

EXAMPLE PROGRAM:

```
import java.awt.*;
import javax.swing.*;

public class Calculator
{
    public Calculator()
    {
        JFrame f=new JFrame("CALCULATOR");
        f.setVisible(true); f.setSize(300,300);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel p=new JPanel();

        JLabel l1=new JLabel("Value A");
        JLabel l2=new JLabel("Value B");
        JLabel l3=new JLabel("Result");

        JTextField t1 = new JTextField(10);
        JTextField t2 = new JTextField(10);
        JTextField t3 = new JTextField(10);

        JButton b1=new JButton("ADD");
        p.add(l1);
        p.add(t1);
        p.add(l2);
        p.add(t2);
        p.add(l3);
        p.add(t3);
        p.add(b1);

        p.setLayout(new GridLayout(4,2));
        f.setContentPane(p);
    }
}
```

```

}

public static void main(String[] args)
{
    Calculator c=new Calculator();
}

}
}

```

4.EXPLAIN IN DETAIL ABOUT LAYOUT MANAGER IN JAVA.

LAYOUT MANAGER:

Layouts are invoked using layout manager interface and set by the setlayout method.

Java LayoutManagers are used to save you the effort of manually putting Components where you want them on a Container.

Even more useful is the fact that a LayoutManager object will handle repositioning your objects whenever the Container is resized.

TYPES OF LAYOUT:

- Flow Layout
- Border Layout
- Grid Layout
- GridBag Layout

FLOWLAYOUT MANAGER

The FlowLayout manager arrange components in its container horizontally left to right and top to bottom starting from center by default.

If you try to add more components in a single row then the row splits into second row.

Justification - FlowLayout.LEFT, FlowLayout.CENTER or FlowLayout.RIGHT.

Default is FlowLayout.CENTER.

The components added to a container using FlowLayout manager, even if you resize the window, component does not change it's original size.

FlowLayout honors the preferred size of the component.

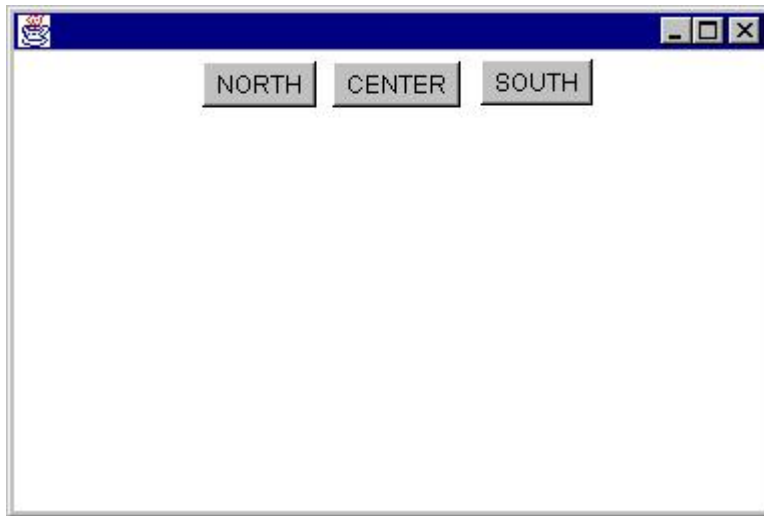
EXAMPLE:

```

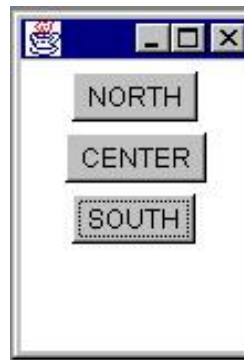
public class TestFlow
{
    public static void main(String args[])
    {
        Panel p = new Panel();
        Button b = new Button("NORTH");
    }
}

```

```
Button b1 = new Button("SOUTH");
Button b2 = new Button("CENTER");
p.add(b);
p.add(b1);
p.add(b2);
Frame f = new Frame();
f.setSize(150,150);
f.setVisible(true);
f.add(p);
    }
}
```

Output:

If you resize the window to smaller size then the top row splits into second row but the the size of the component does not change, which means Flow Layout manager honors the component preferred size.

**BORDERLAYOUT MANAGER**

The BorderLayout manager is a default layout manager for frames.

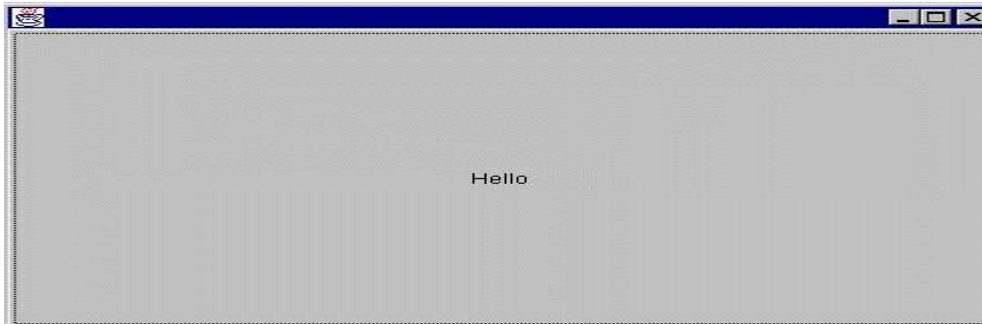
This layout manager divides container in to five regions where you can place components, called "North", "South", "East", "West", and "Center".

The center region occupiees leftover space if you did not place any component in other regions.

Example :

```
import java.awt.*;
public class TestBorder
{
public static void main(String args[])
{
    Button b = new Button("Hello");
    Frame f= new Frame();
    f.setSize(150,150);
    f.setVisible(true);
    f.add(b);
}
}
```

Output:



If you don't specify the location of the component, then the center is the default location of the component in the BorderLayout, and component occupiees all the leftover space.

While adding a component you can use "North", "South", "East", "West", "Center" or the following constants to specify the location of the component. You can also specify mixing of both.

BorderLayout.NORTH

BorderLayout.SOUTH

BorderLayout.EAST

BorderLayout.WEST

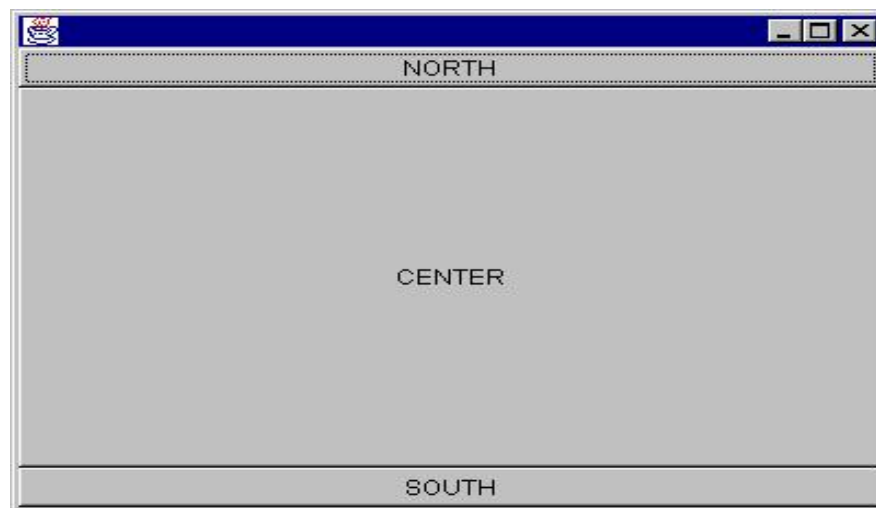
BorderLayout.CENTER

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

```

    Button b = new Button("NORTH");
    Button b1 = new Button("SOUTH");
    Button b2 = new Button("CENTER");
    Frame f= new Frame();
    f.setSize(150,150);
    f.setVisible(true);
    f.add(b, BorderLayout.NORTH);
    f.add(b1, "South");
    f.add(b2, BorderLayout.CENTER);
}
}

```

Output:

If you use BorderLayout when you resize the window, the components in the container also resized.

GRIDLAYOUT MANAGER

The GridLayout manager divides the space in the container in to ROWS AND COLUMNS specified in the constructor.

Basically it divides the region into number of rows and columns which you specified while setting the layout manager for the container.

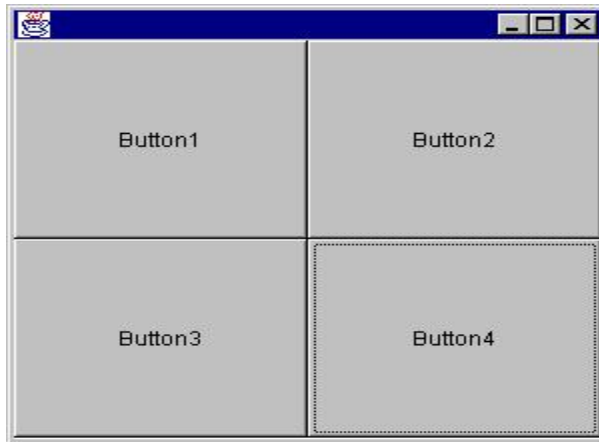
Example:

```

public class TestGrid {
    public static void main(String args[])
    { Panel p = new Panel();
      Button b1 = new Button("Button1");
      Button b2 = new Button("Button2");
      Button b3 = new Button("Button3");
    }
}

```

```
Button b4 = new Button("Button4");
Frame f = new Frame();
f.setSize(150,150);
f.setVisible(true);
f.add(p);
p.setLayout(new GridLayout(2,2));
p.add(b1 );
p.add(b2);
p.add(b3);
p.add(b4);
}
}
```

Output:

If you use GridLayout manager and when you resize the window all the components in the container also resized.

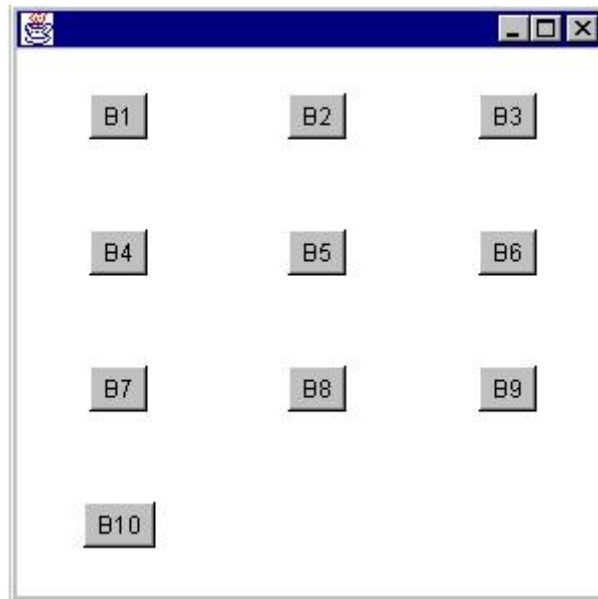
GRIDBAG LAYOUT MANAGER

GridBagLayout determines the number of rows and columns from the constraints placed upon the components it lays out.

It does not require providing the rows and columns as in the GridLayout.

It allows spanning the components to more than one grid cell.

Output:



5. JAVA GRAPHICS CLASS IN DETAIL

Graphics:

Java supports graphics along with the AWT. A graphics context is encapsulated by the graphics class and obtained in 2 ways

It is passed to an applet when one of its various methods such as `paint()` and `update()` is called. It is returned by the `getGraphics()` method of component.

Working with drawing:

Graphic class defines a number of drawing function. Each shape can be drawn edge-only or filled.

Methods are:

drawLine

Draws a line between the coordinates $(x1,y1)$ and $(x2,y2)$. The line is drawn below and to the left of the logical coordinates

```
public abstract void drawLine(int x1,int y1, int x2, int y2)
```

Parameters:

- x1 - the first point's x coordinate
- y1 - the first point's y coordinate
- x2 - the second point's x coordinate
- y2 - the second point's y coordinate

drawRect

Draws the outline of the specified rectangle using the current color. Use `drawRect(x, y, width-1, height-1)` to draw the outline inside the specified rectangle.

```
public void drawRect(int x, int y, int width, int height)
```

Parameters:

x - the x coordinate
y - the y coordinate
width - the width of the rectangle
height - the height of the rectangle

fillRect

Fills the specified rectangle with the current color.

```
public abstract void fillRect(int x, int y, int width, int height)
```

Parameters:

x - the x coordinate
y - the y coordinate
width - the width of the rectangle
height - the height of the rectangle

Other methods are

- **draw3DRect**(int, int, int, int, boolean)
Draws a highlighted 3-D rectangle.
- **drawArc**(int, int, int, int, int, int)
Draws an arc bounded by the specified rectangle from `startAngle` to `endAngle`.
- **drawBytes**(byte[], int, int, int, int)
Draws the specified bytes using the current font and color.
- **drawChars**(char[], int, int, int, int)
Draws the specified characters using the current font and color.
- **drawImage**(Image, int, int, ImageObserver)
Draws the specified image at the specified coordinate (x, y).
- **drawImage**(Image, int, int, int, int, ImageObserver)
Draws the specified image inside the specified rectangle.
- **drawOval**(int, int, int, int)
Draws an oval inside the specified rectangle using the current color.
- **drawPolygon**(int[], int[], int)
Draws a polygon defined by an array of x points and y points.
- **drawPolygon**(Polygon)
Draws a polygon defined by the specified point.
- **drawRoundRect**(int, int, int, int, int, int)
Draws an outlined rounded corner rectangle using the current color.
- **drawString**(String, int, int)

Draws the specified String using the current font and color.

- **fill3DRect**(int, int, int, int, boolean)
Paints a highlighted 3-D rectangle using the current color.
- **fillArc**(int, int, int, int, int)
Fills an arc using the current color.
- **fillOval**(int, int, int, int)
Fills an oval inside the specified rectangle using the current color.
- **fillPolygon**(int[], int[], int)
Fills a polygon with the current color.
- **fillPolygon**(Polygon)
Fills the specified polygon with the current color.
- **fillRoundRect**(int, int, int, int, int, int)
Draws a rounded rectangle filled in with the current color.

WORKING WITH COLORS:

Color can be given to the shapes drawn using draw methods.

Methods are

getColor

Gets the current color

```
public abstract Color getColor()
```

setColor

```
public abstract void setColor(Color c)
```

Sets the current color to the specified color. All subsequent graphics operations will use this specified color.

Parameters:

c - the color to be set

setPaintMode

Sets the default paint mode to overwrite the destination with the current color.

```
public abstract void setPaintMode()
```

WORKING WITH FONTS:

getFont

Gets the current font.

```
public abstract Font getFont()
```

setFont

Sets the font for all subsequent text-drawing operations.

```
public abstract void setFont(Font font)
```

Parameters:

font - the specified font

getFontMetrics

Gets the current font metrics.

```
public FontMetrics getFontMetrics()
```

getFontMetrics

Gets the current font metrics for the specified font.

```
public abstract FontMetrics getFontMetrics(Font f)
```

Parameters:

f - the specified font

Some of the other methods in graphics are

clearRect(int, int, int, int)

Clears the specified rectangle by filling it with the current background color of the current drawing surface.

clipRect(int, int, int, int)

Clips to a rectangle.

copyArea(int, int, int, int, int, int)

Copies an area of the screen.

create()

Creates a new Graphics Object that is a copy of the original Graphics Object.

create(int, int, int, int)

Creates a new Graphics Object with the specified parameters, based on the original Graphics Object.

dispose()

Disposes of this graphics context.

scale(float, float)

Scales the graphics context.

setColor(Color)

Sets the current color to the specified color.

setXORMode(Color)

Sets the paint mode to alternate between the current color and the new specified color.

toString()

Returns a String object representing this Graphic's value.

translate(int, int)

Translates the specified parameters into the origin of the graphics context.

6. DESCRIBE ABOUT EVENT HANDLING MECHANISM

Event describes the change of state of any object.

Example : Pressing a button, Entering a character in Textbox.

Any action that user performs on a GUI component must be listened and necessary action should be taken.

For example, if a user clicks on a *Exit* button, then we need to write code to exit the program.

we need to know that the user has clicked the button. This process of knowing is called as listening . The action done by the user is called an event. Writing the corresponding code for a user action is called as Event handling

COMPONENTS OF EVENT HANDLING

Event handling has three main components,

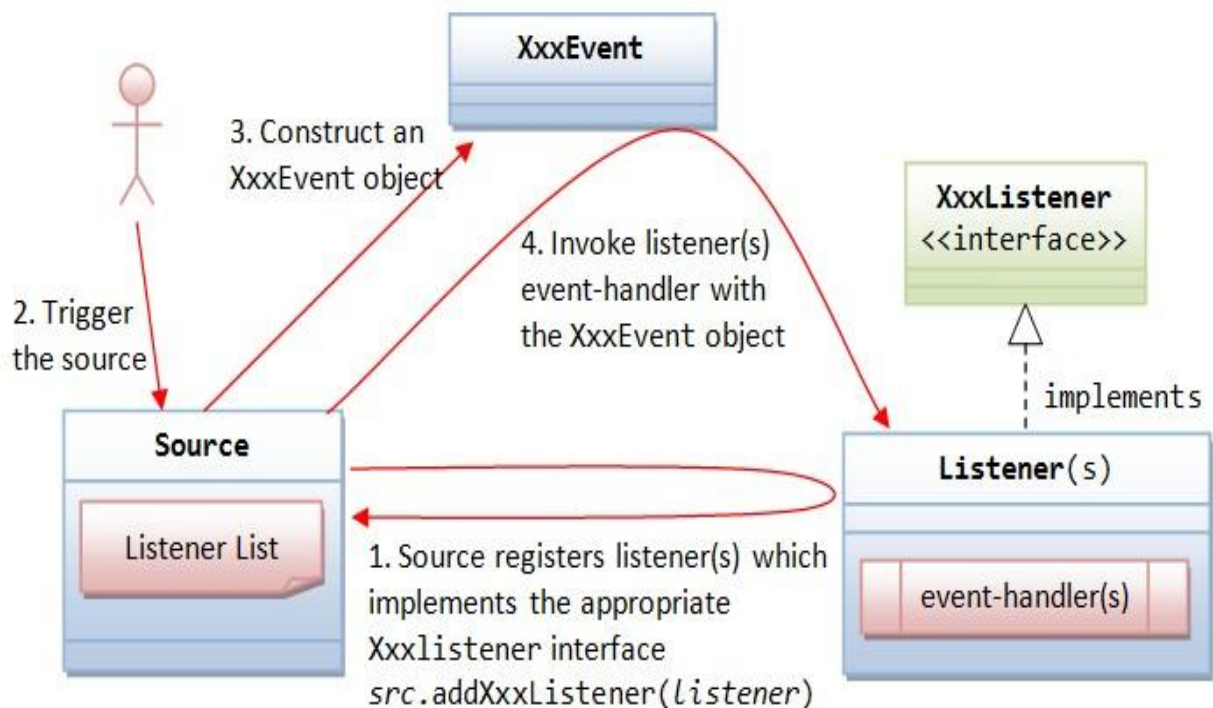
- Events : An event is a change of state of an object.
- Events Source : Event source is an object that generates an event.
- Listeners : A listener is an object that listens to the event. A listener gets notified when an event occurs

STEPS TO PERFORM EVENT HANDLING

Following steps are required to perform event handling:

- Implement the Listener interface and overrides its methods
- Register the component with the Listener

WORKING OF EVENT HANDLING



THE SEQUENCE OF STEPS IS ILLUSTRATED ABOVE:

1. *The source object registers its listener(s) for a certain type of event.*

Source object fires event event upon triggered. For example, clicking an Button fires an ActionEvent, mouse-click fires MouseEvent, key-type fires KeyEvent, etc.

The source and listener understand each other via an agreed-upon interface.

For example, if a source is capable of firing an event called XxxEvent (e.g., MouseEvent) involving various operational modes (e.g., mouse-clicked, mouse-entered, mouse-exited, mouse-pressed, and mouse-released).

Firstly, we need to declare an interface called XxxListener (e.g., MouseListener) containing the names of the handler methods. Recall that an interface contains only abstract methods without implementation.

For example,

```
// A MouseListener interface, which declares the signature of the handlers
// for the various operational modes.
public interface MouseListener {
    public void mousePressed(MouseEvent evt); // Called back upon mouse-button pressed
    public void mouseReleased(MouseEvent evt); // Called back upon mouse-button released
    public void mouseClicked(MouseEvent evt); // Called back upon mouse-button clicked
    (pressed and released)
    public void mouseEntered(MouseEvent evt); // Called back when mouse pointer entered
    the component
    public void mouseExited(MouseEvent evt); // Called back when mouse pointer exited the
    component
}
```

Secondly, all the listeners interested in the XxxEvent must implement the XxxListener interface. That is, the listeners must provide their own implementations (i.e., programmed responses) to all the abstract methods declared in the XxxListener interface. In this way, the listener(s) can respond to these events appropriately.

For example

```
// An example of MouseListener, which provides implementation to the handler methods
class MyMouseListener implements MouseListener {
    @Override
    public void mousePressed(MouseEvent e)
    { System.out.println("Mouse-button
    pressed!");
    }

    @Override
    public void mouseReleased(MouseEvent e)
    { System.out.println("Mouse-button
    released!");
    }

    @Override
    public void mouseClicked(MouseEvent e)
    { System.out.println("Mouse-button clicked (pressed and
    released!");
    }
}
```

```

}

@Override
public void mouseEntered(MouseEvent e)
{ System.out.println("Mouse-pointer entered the source
component!");
}

@Override
public void mouseExited(MouseEvent e)
{ System.out.println("Mouse exited-pointer the source
component!");
}
}
}

```

Thirdly, in the source, we need to maintain a list of listener object(s), and define two methods: `addXxxListener()` and `removeXxxListener()` to add and remove a listener from this list.

The signature of the methods are:

```

public void addXxxListener(XxxListener l);
public void removeXxxListener(XxxListener l);

```

In summary, we identify the source, the event-listener interface, and the listener object. The listener must implement the event-listener interface. The source object then registers listener object via the `addXxxListener()` method:

```

aSource.addXxxListener(aListener); // aSource registers aListener for XxxEvent

```

2. The source is triggered by a user.
3. The source create an `XxxEvent` object, which encapsulates the necessary information about the activation. For example, the (x, y) position of the mouse pointer, the text entered, etc.
4. Finally, for each of the listeners in the listener list, the source invokes the appropriate handler on the listener(s), which provides the programmed response.

In brief, *triggering a source fires an event to all its registered listeners, and invoke an appropriate handler of the listener.*

INTERFACE	INTERFACE METHODS	ADD METHOD	EVENT CLASS
ActionListener	actionPerformed (ActionEvent)	addActionListener()	ActionEvent
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)	addAdjustmentListener()	AdjustmentEvent
ComponentListener	componentHidden(ComponentEvent)	addComponentListener()	ComponentEvent
	componentMoved(ComponentEvent)		
	componentResized(ComponentEvent)		
	componentShown(ComponentEvent)		
ContainerListener	componentAdded(ComponentEvent)	addContainerListener()	ContainerEvent
	componentRemoved(ComponentEvent)		
FocusListener	focusGained(FocusEvent)	addFocusListener()	FocusEvent
	focusLost(FocusEvent)		

ItemListener	itemStateChanged(ItemEvent)	addItemListener()	ItemEvent
KeyListener	keyPressed(KeyEvent)	addKeyListener()	KeyEvent
	keyReleased(KeyEvent)		
	keyTyped(KeyEvent)		
MouseListener	mouseClicked(MouseEvent)	addMouseListener()	MouseEvent
	mouseEntered(MouseEvent)		
	mouseExited(MouseEvent)		
	mousePressed(MouseEvent)		
	mouseReleased(MouseEvent)		

MouseListener	mouseDragged(MouseEvent)	addMouseListener()	MouseEvent
	mouseMoved(MouseEvent)		
Text:Listener	textValueChanged(TextEvent)	addText:Listener()	TextEvent
WindowListener	windowActivated(WindowEvent)	addWindowListener()	WindowEvent
	windowClosed(WindowEvent)		
	windowClosing(WindowEvent)		
	windowDeactivated(WindowEvent)		
	windowDeiconified(WindowEvent)		
	windowIconified(WindowEvent)		
	windowOpened(WindowEvent)		

UNIT 4 - 2 MARKS**1. DEFINE GENERICS**

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects. Before generics, we can store any type of objects in collection i.e. non-generic. Now generics, forces the java programmer to store specific type of objects

2. STATE THE ADVANTAGES OF GENERICS

There are mainly 3 advantages of generics. They are as follows:

1) Type-safety: We can hold only a single type of objects in generics. It doesn't allow to store other objects.

2) Type casting is not required: There is no need to typecast the object.

Before Generics, we need to type cast.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
```

After Generics, we don't need to typecast the object.

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

3) Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

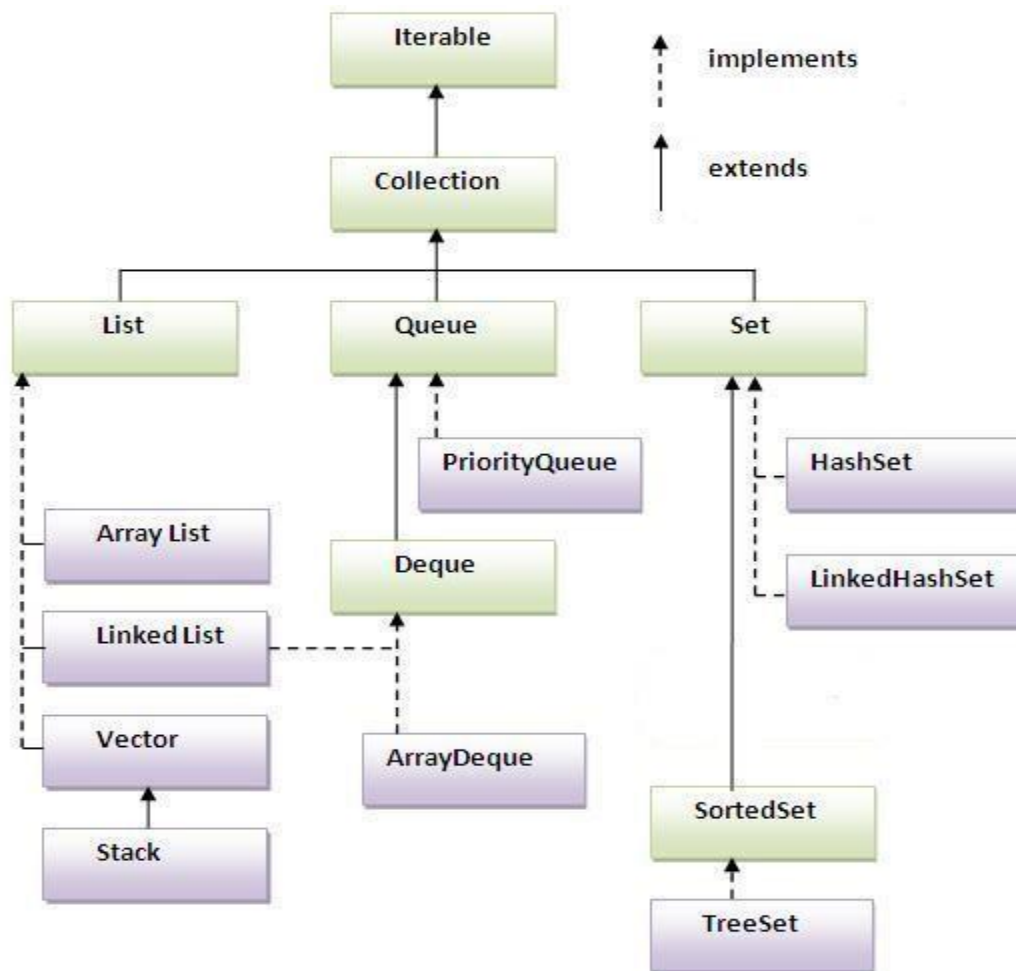
```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);//Compile Time Error
```

3. DEFINE COLLECTIONS FRAMEWORK

Collection framework represents a unified architecture for storing and manipulating group of object.

All the operations that you perform on a data such as searching, sorting, insertion, deletion etc. can be performed by Java Collection Framework

4. STATE THE COLLECTIONS HIERARCHY



5. DEFINE SET INTERFACE

The Set interface is used to represent a group of unique elements. It extends the Collection interface. The class HashSet, LinkedHashSet implements the Set interface.

6. DEFINE LIST INTERFACE

The list interface extends the Collection interface to represent sequence of numbers in a fixed order with allowing duplicate elements. Classes ArrayList, Vector and LinkedList are implementation of List interface

7. DEFINE MAP INTERFACE

The Map Interface is a basic interface that is used to represent mapping of keys to values. A map contains values based on the key i.e. key and value pair. Each pair is known as an entry. Map contains only unique elements. Classes HashMap and LinkedHashMap are implementations of Map interface

8. WHAT IS ITERATOR?

Iterator interface provides the facility of iterating the elements in forward direction only

Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

1. **public boolean hasNext()** it returns true if iterator has more elements.

2. **public Object next()** it returns the element and moves the cursor pointer to the next element
3. **public void remove()** it removes the last elements returned by the iterator. It is rarely used

9. LIST THE CLASS COMES UNDER LIST

ARRAYLIST
LINKEDLIST
VECTOR

10. LIST THE CLASS COMES UNDER SET

HASHSET
LINKEDHASHSET
TREESET

11. LIST THE CLASS COMES UNDER MAP

HASHMAP
LINKEDHASHMAP
TREEMAP

12. STATE THE COMMON METHODS IN COLLECTIONS

No.	METHOD	DESCRIPTION
1	add(Object element)	is used to insert an element in this collection.
2	addAll(Collection c)	is used to insert the specified collection elements in the invoking collection.
3	remove(Object element)	is used to delete an element from this collection.
4	removeAll(Collection c)	is used to delete all the elements of specified collection from the invoking collection.
5	retainAll(Collection c)	is used to delete all the elements of invoking collection except the specified collection.

13. COMPARE LIST AND SET

No.	LIST	SET
1	ALLOW DUPLICATE ELEMENTS	DON'T ALLOW DUPLICATE ELEMENTS
2	ORDER IS MAINTAINED	ORDER IS NOT MAINTAINED
3	USE LISTITERATOR INTERFACE	USE ITERATOR INTERFACE
4	CLASSES: ARRAYLIST, LINKEDLIST	CLASSES: HASHSET, TREESET

14.DISTINGUISH BETWEEN ARRAYLIST AND LINKEDLIST IN COLLECTIONS

No.	ARRAYLIST	LINKEDLIST
1	USES A DYNAMIC ARRAY FOR STORING THE ELEMENTS	USES DOUBLY LINKED LIST TO STORE THE ELEMENTS
2	RANDOM ACCESS	NO RANDOM ACCESS
3	MANIPULATION SLOW BECAUSE A LOT OF SHIFTING NEEDS TO BE OCCURRED	MANIPULATION FAST BECAUSE NO SHIFTING NEEDS TO BE OCCURRED

15.LIST SOME OF THE CLASSES IN UTILITY PACKAGE

Arrays
Date
Calendar
Dictionary
Currency

17.STATE CALENDER CLASS

The Calendar class is an abstract class that provides methods for converting between a specific instant in time and a set of **calendar fields** such as YEAR, MONTH, DAY_OF_MONTH, HOUR, and so on, and for manipulating the calendar fields, such as getting the date of the next week

16.WHAT IS SCANNER CLASS

A simple text scanner which can parse primitive types and strings using regular expressions

17.DEFINE INNER CLASS

Java inner class or nested class is a class i.e. declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods

18.STATE THE ADVANTAGES OF INNER CLASS

There are basically three advantages of inner classes in java. They are as follows:

- 1) Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.
- 2) Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.
- 3) **Code Optimization:** It requires less code to write.

19.WHAT ARE THE TYPES OF INNER CLASSES IN JAVA

- a)Member inner class
- b)Anonymous inner class
- c)Local inner class

20. WHAT IS STATIC NESTED CLASS

A static class created within class is called as static nested class. Static nested class is not an inner class

21. DEFINE JAVA LOCAL INNER CLASS

A class i.e. created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method

22. DEFINE JAVA MEMBER INNER CLASS

A non-static class that is created inside a class but outside a method is called member inner class.

23. DEFINE JAVA ANONYMOUS INNER CLASS

A class that has no name is known as an anonymous inner class in java. It should be used if you have to override a method of a class or interface. Java Anonymous inner class can be created by two ways:

1. Class (may be abstract or concrete).
2. Interface

24. HOW JAVA COMMUNICATE WITH THE DATABASE

JDBC is a standard interface for connecting to databases from Java. Java API for connecting programs written in Java to the data in relational databases. The JDBC classes and interfaces are in the *java.sql* package

25. LIST THE STEPS TO FOLLOW IN JDBC

There are 7 steps to follow in JDBC

1. Import package java.sql
2. Register the driver
3. Establish the connection
4. Create the statement
5. Execute the query
6. Process the result
7. Close the connection

26. WHAT IS JDBC DRIVER

A JDBC Driver is an interpreter that translates JDBC method calls to vendor-specific database commands

Implements interfaces in java.sql

Can also provide a vendor's extensions to the JDBC standard

27. WHAT ARE THE TYPES OF DRIVER IN JDBC

There are mainly four types

- JDBC-ODBC
- Native API Drivers
- Network API
- Network Protocol Driver

28. WHAT ARE THE TWO WAYS USED TO REGISTER DRIVER IN JDBC

There are Two Ways:

1) Using `Class.forName()`

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

2) Using `DriverManager.registerDriver()`

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
```

29. WHAT IS STATEMENT CLASS

A Statement object sends your SQL statement to the database.

You need an active connection to create a JDBC statement.

Statement has three methods to execute a SQL statement:

1. `executeQuery()` for QUERY statements
 2. `executeUpdate()` for INSERT, UPDATE, DELETE, or DDL statements
- `execute()` for either type of statement

30. DEFINE PREPARED STATEMENT IN JDBC

A PreparedStatement object holds precompiled SQL statements.

Use this object for statements you want to execute more than once.

A prepared statement can contain variables that you supply each time you execute the statement

31. WHAT ARE THE TYPES OF QUERIES IN DATABASE

There are mainly four queries in database management. They are

Insert

Select

Update

Delete

32. DEFINE JAVA SECURITY

The Java platform provides a number of features designed to improve the **security** of Java applications. This includes enforcing runtime constraints through the use of the Java Virtual Machine (JVM), a security manager that sandboxes untrusted code from the rest of the operating system, and a suite of security APIs that Java developers can utilize

33. WHAT IS SAND BOX SECURITY MODEL

The sandbox security model is an intrinsic part of Java's architecture. The sandbox, a shell that surrounds a running Java program, protects the host system from malicious code. This security model helps give users confidence in downloading untrusted code across network.

34. WHAT ARE THE SAFETY FEATURES BUILT INTO THE JVM

Type-safe reference casting

Structured memory access (no pointer arithmetic)

Automatic garbage collection (can't explicitly free allocated memory)

Array bounds checking

Checking references for null

35.WHAT IS DRIVERMANAGER?

The basic service to manage set of JDBC drivers.

36.WHAT IS CLASS.FORNAME() DOES AND HOW IT IS USEFUL?

It loads the class into the ClassLoader. It returns the Class. Using that you can get the instance (`–class-instance||.newInstance()`).

37.WHAT ARE BYTE STREAM IN JAVA?

The byte stream classes provide a rich environment for handling byte-oriented I/O.

List of Byte Stream classes

- ByteArrayInputStream
- ByteArrayOutputStream
- FilteredByteStreams
- BufferedByteStreams

38.WHAT ARE CHARACTER STREAM IN JAVA?

The Character Stream classes provide a rich environment for handling character-oriented I/O.

List of Character Stream classes

- FileReader
- FileWriter
- CharArrayReader
- CharArrayWriter

39.WRITE A NOTE ON CHAR ARRAY READER

The CharArrayReader allows the usage of a character array as an InputStream. The usage of CharArrayReader class is similar to ByteArrayInputStream. The constructor is given below:

```
public CharArrayReader(char c[ ])
```

40.HOW WILL YOU FIND OUT THE LENGTH OF A STRING IN JAVA? GIVE AN EXAMPLE?

length() method is used to number of characters is string. For example,

```
String str="Hello";
System.out.println("Length of string is "+str.length( ));
```


UNIT - IV**11 MARKS****1. WRITE ABOUT COLLECTION CLASSES IN JAVA**

Collection Framework provides an architecture to store and manipulate the group of objects.

All the operations that you perform on a data such as searching, sorting, insertion, deletion etc. can be performed by Java Collection Framework.

Collection simply means a single unit of objects. Collection framework provides many interfaces (Set, List, Queue, Deque etc.) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet etc).

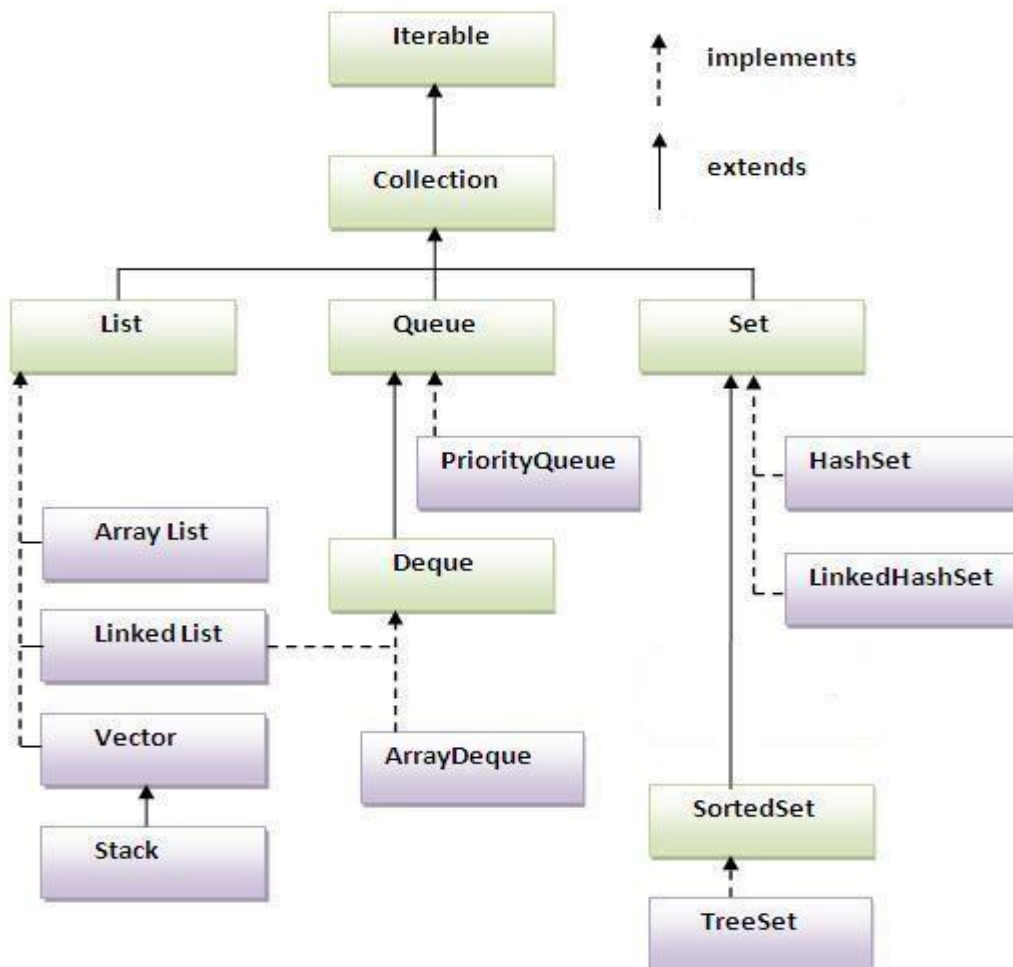
Collection framework represents a unified architecture for storing and manipulating group of object.

It has:

1. Interfaces and its implementations i.e. classes
2. Algorithm

HIERARCHY OF COLLECTION FRAMEWORK

The **java.util** package contains all the classes and interfaces for Collection framework.



METHODS OF COLLECTION INTERFACE

There are many methods declared in the Collection interface. They are as follows:

No.	METHOD	DESCRIPTION
1	add(Object element)	is used to insert an element in this collection.
2	addAll(Collection c)	is used to insert the specified collection elements in the invoking collection.
3	remove(Object element)	is used to delete an element from this collection.
4	removeAll(Collection c)	is used to delete all the elements of specified collection from the invoking collection.
5	retainAll(Collection c)	is used to delete all the elements of invoking collection except the specified collection.
6	size()	return the total number of elements in the collection.
7	clear()	removes the total no of element from the collection.
8	contains(object element)	is used to search an element.
9	containsAll(Collection c)	is used to search the specified collection in this collection.
10	iterator()	returns an iterator.
11	toArray()	converts collection into array.
12	isEmpty()	checks if collection is empty.
13	equals(Object element)	matches two collection.
14	hashCode()	returns the hashcode number for collection.

ITERATOR INTERFACE

Iterator interface provides the facility of iterating the elements in forward direction only

Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

1. **public boolean hasNext()** it returns true if iterator has more elements.
2. **public Object next()** it returns the element and moves the cursor pointer to the next element
3. **public void remove()** it removes the last elements returned by the iterator. It is rarely used.

LIST INTERFACE

The list interface extends the Collection interface to represent sequence of numbers in a fixed order with allowing duplicate elements. Classes ArrayList, Vector and LinkedList are implementation of List interface.

List Implemented Classes**ArrayList**

- uses a dynamic array for storing the elements. It extends AbstractList class and implements List interface.
- can contain duplicate elements.
- maintains insertion order.
- not synchronized.
- random access because array works at the index basis.
- manipulation slow because a lot of shifting needs to be occurred.

LinkedList

- uses doubly linked list to store the elements. It extends the AbstractList class and implements List and Deque interfaces.
- can contain duplicate elements.
- maintains insertion order.
- not synchronized.
- No random access.
- manipulation fast because no shifting needs to be occurred.
- can be used as list, stack or queue

EXAMPLE

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

public class arraylist
{
    public static void main(String args[])
    {
        ArrayList al = new ArrayList(); //Creating object for ArrayList

        System.out.println("Array List");

        al.add("A");           //Adding element on ArrayList
        al.add("B");
        al.add("C");
        al.add("D");

        System.out.println("Size of Array List: " + al.size());

        System.out.println("Contents of al: " + al); //Displaying element on ArrayList
    }
}
```

```

al.add(1, "X");           //Adding element at position 1 on ArrayList

System.out.println("Contents of al: " + al); //Displaying element on ArrayList

al.remove("C");         //Removing element on ArrayList

System.out.println("Contents of al: " + al);

al.remove(2); //Removing element at position 2 on ArrayList

System.out.println("Contents of al: " + al);

//Finding element at position 3 on ArrayList
System.out.println("Element at position 3: " + al.get(2));

//Finding position of element on ArrayList
System.out.println("Position of character X is : " + al.indexOf("X"));

if(al.contains("S")) //Searching element on ArrayList
{
    System.out.println("Element Found");
}
else
{
    System.out.println("Element Not Found");
}

al.clear();           //Clearing all element on ArrayList

System.out.println("Size of Array List: " + al.size());
}
}

```

SET INTERFACE

The Set interface is used to represent a group of unique elements. It extends the Collection interface. The class HashSet, LinkedHashSet implements the Set interface.

Set Implemented Classes

HashSet

uses hashtable to store the elements. It extends AbstractSet class and implements Set interface.

contains unique elements only.

LinkedHashSet

contains unique elements only like HashSet. It extends HashSet class and implements Set interface.

maintains insertion order.

SORTEDSET INTERFACE

The SortedSet interface extends the Set interface. It provides extra functionality of keeping the elements sorted. So SortedSet interface is used to represent collections consisting of unique, sorted elements. The class TreeSet is an implementation of interface SortedSet.

Sortedset Implemented Class**Treeset**

contains unique elements only like HashSet. The TreeSet class implements NavigableSet interface that extends the SortedSet interface.

maintains ascending order

MAP INTERFACE

The Map Interface is a basic interface that is used to represent mapping of keys to values. A map contains values based on the key i.e. key and value pair. Each pair is known as an entry. Map contains only unique elements. Classes HashMap and LinkedHashMap are implementations of Map interface

Commonly Used Methods of Map Interface

1. **public Object put(object key, Object value):** is used to insert an entry in this map.
2. **public void putAll(Map map):** is used to insert the specified map in this map.
3. **public Object remove(object key):** is used to delete an entry for the specified key.
4. **public Object get(Object key):** is used to return the value for the specified key.
5. **public boolean containsKey(Object key):** is used to search the specified key from this map.
6. **public boolean containsValue(Object value):** is used to search the specified value from this map.
7. **public Set keySet():** returns the Set view containing all the keys.
8. **public Set entrySet():** returns the Set view containing all the keys and values.

Map Implemented Classes**HashMap**

A HashMap contains values based on the key. It implements the Map interface and extends AbstractMap class.

It contains only unique elements.

It may have one null key and multiple null values.

It maintains no order

LinkedHashMap

A LinkedHashMap contains values based on the key. It implements the Map interface and extends HashMap class.

It contains only unique elements.

It may have one null key and multiple null values.

It is same as HashMap instead maintains insertion order.

SORTED MAP INTERFACE

The SortedMap Interface extends Map interface and maintains their mappings in key order. The class TreeMap implements SortedMap interface

SortedMap Implemented Class

TreeMap

A TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.

It contains only unique elements.

It cannot have null key but can have multiple null values.

It is same as HashMap instead maintains ascending order

EXAMPLE PROGRAM

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

public class javacollections
{
    public static void main(String[] args)
    {
        List linked = new LinkedList();
        System.out.println("\n*****LinkedList class*****");
        linked.add("element1");
        linked.add("element2");
        System.out.println(linked);

        List array = new ArrayList();
        System.out.println("\n*****ArrayList class*****");
        array.add("x");
        array.add("y");
        System.out.println(array);

        Set hashSet = new HashSet();
        System.out.println("\n*****HashSet class*****");
        hashSet.add("set1");
        hashSet.add("set2");
        System.out.println(hashSet);

        SortedSet treeSet = new TreeSet();
        System.out.println("\n*****TreeSet class*****");
        treeSet.add("1");
        treeSet.add("2");
        System.out.println(treeSet);

        LinkedHashSet linkedHashset = new LinkedHashSet();
        System.out.println("\n*****LinkedHashSet class*****");
        linkedHashset.add("one");
        linkedHashset.add("two");
```

```

System.out.println(linkedHashSet);

Map map1 = new HashMap();
System.out.println("\n*****Hashmap class*****");
map1.put("key1", "J");
map1.put("key2", "K");
System.out.println(map1.keySet());
System.out.println(map1.values());
System.out.println(map1);
    }
}

```

2. EXPLAIN IN DETAIL ABOUT THE UTILITY PACKAGE

The java.util package defines a number of useful classes, primarily collections classes that are useful for working with groups of objects. This package should not be considered merely a utility package that is separate from the rest of the language; in fact, Java depends directly on several of the classes in this package. Figure 1 shows the collection classes of this package, while Figure 2 shows the other classes.

It contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

LIST OF INTERFACES

INTERFACES	DESCRIPTION
Collection <E>	The root interface in the <i>collection hierarchy</i> .
Comparator <T>	A comparison function, which imposes a <i>total ordering</i> on some collection of objects.
Deque <E>	A linear collection that supports element insertion and removal at both ends.
Enumeration <E>	An object that implements the Enumeration interface generates a series of elements, one at a time.
EventListener	A tagging interface that all event listener interfaces must extend.
Formattable	The Formattable interface must be implemented by any class that needs to perform custom formatting using the 's' conversion specifier of Formatter .
Iterator <E>	An iterator over a collection.
List <E>	An ordered collection (also known as a <i>sequence</i>).
ListIterator <E>	An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.
Map <K,V>	An object that maps keys to values.
Map.Entry <K,V>	A map entry (key-value pair).
NavigableMap <K,V>	A SortedMap extended with navigation methods returning the closest matches for given search targets.
NavigableSet <E>	A SortedSet extended with navigation methods reporting closest matches for given search targets.
Observer	A class can implement the Observer interface when it wants to be informed of changes in observable objects.

Queue <E>	A collection designed for holding elements prior to processing.
RandomAccess	Marker interface used by List implementations to indicate that they support fast (generally constant time) random access.
Set <E>	A collection that contains no duplicate elements.
SortedMap <K,V>	A Map that further provides a <i>total ordering</i> on its keys.
SortedSet <E>	A Set that further provides a <i>total ordering</i> on its elements.

LIST OF CLASSES

CLASS	DESCRIPTION
AbstractCollection <E>	This class provides a skeletal implementation of the Collection interface, to minimize the effort required to implement this interface.
AbstractList <E>	This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array).
AbstractMap <K,V>	This class provides a skeletal implementation of the Map interface, to minimize the effort required to implement this interface.
AbstractMap.SimpleEntry <K,V>	An Entry maintaining a key and a value.
AbstractMap.SimpleImmutableEntry <K,V>	An Entry maintaining an immutable key and value.
AbstractQueue <E>	This class provides skeletal implementations of some Queue operations.
AbstractSequentialList <E>	This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "sequential access" data store (such as a linked list).
AbstractSet <E>	This class provides a skeletal implementation of the Set interface to minimize the effort required to implement this interface.
ArrayDeque <E>	Resizable-array implementation of the Deque interface.
ArrayList <E>	Resizable-array implementation of the List interface.
Arrays	This class contains various methods for manipulating arrays (such as sorting and searching).
BitSet	This class implements a vector of bits that grows as needed.
Calendar	The Calendar class is an abstract class that provides methods for converting between a specific instant in time and a set of calendar fields such as YEAR, MONTH, DAY_OF_MONTH, HOUR, and so on, and for manipulating the calendar fields, such as getting the date of the next week.
Collections	This class consists exclusively of static methods that operate on or return collections.
Currency	Represents a currency.
Date	The class Date represents a specific instant in time, with millisecond precision.
Dictionary <K,V>	The Dictionary class is the abstract parent of any class, such as Hashtable, which maps keys to values.
EnumMap <K extends Enum<K>,V>	A specialized Map implementation for use with enum type keys.

>	
EnumSet <E extends Enum <E>>	A specialized Set implementation for use with enum types.
EventListenerProxy <T extends EventListen er >	An abstract wrapper class for an EventListener class which associates a set of additional parameters with the listener.
EventObject	The root class from which all event state objects shall be derived.
FormattableFlags	FomattableFlags are passed to the Formattable.formatTo() method and modify the output format for Formattables .
Formatter	An interpreter for printf-style format strings.
GregorianCalendar	GregorianCalendar is a concrete subclass of Calendar and provides the standard calendar system used by most of the world.
HashMap <K,V>	Hash table based implementation of the Map interface.
HashSet <E>	This class implements the Set interface, backed by a hash table (actually a HashMap instance).
Hashtable <K,V>	This class implements a hash table, which maps keys to values.
IdentityHashMap <K ,V>	This class implements the Map interface with a hash table, using reference-equality in place of object-equality when comparing keys (and values).
LinkedHashMap <K, V>	Hash table and linked list implementation of the Map interface, with predictable iteration order.
LinkedHashSet <E>	Hash table and linked list implementation of the Set interface, with predictable iteration order.
LinkedList <E>	Doubly-linked list implementation of the List and Deque interfaces.
ListResourceBundl e	ListResourceBundle is an abstract subclass of ResourceBundle that manages resources for a locale in a convenient and easy to use list.
Locale	A Locale object represents a specific geographical, political, or cultural region.
Locale.Builder	Builder is used to build instances of Locale from values configured by the setters.
Objects	This class consists of static utility methods for operating on objects.
Observable	This class represents an observable object, or "data" in the model-view paradigm.
PriorityQueue <E>	An unbounded priority queue based on a priority heap.
Properties	The Properties class represents a persistent set of properties.
PropertyPermissio n	This class is for property permissions.
PropertyResourceB undle	PropertyResourceBundle is a concrete subclass of ResourceBundle that manages resources for a locale using a set of static strings from a property file.
Random	An instance of this class is used to generate a stream of pseudorandom numbers.
ResourceBundle	Resource bundles contain locale-specific objects.
ResourceBundle.Co ntrol	ResourceBundle.Control defines a set of callback methods that are invoked by the ResourceBundle.getBundle factory methods during the bundle loading process.

Scanner	A simple text scanner which can parse primitive types and strings using regular expressions.
ServiceLoader<S>	A simple service-provider loading facility.
SimpleTimeZone	SimpleTimeZone is a concrete subclass of TimeZone that represents a time zone for use with a Gregorian calendar.
Stack<E>	The Stack class represents a last-in-first-out (LIFO) stack of objects.
StringTokenizer	The string tokenizer class allows an application to break a string into tokens.
Timer	A facility for threads to schedule tasks for future execution in a background thread.
TimerTask	A task that can be scheduled for one-time or repeated execution by a Timer.
TimeZone	TimeZone represents a time zone offset, and also figures out daylight savings.
TreeMap<K,V>	A Red-Black tree based NavigableMap implementation.
TreeSet<E>	A NavigableSet implementation based on a TreeMap .
UUID	A class that represents an immutable universally unique identifier (UUID).

3. DESCRIBE ABOUT THE ARRAYS CLASS IN UTILITY PACKAGE?

This class defines static methods for sorting, searching, and performing other useful operations on arrays. It also defines the `asList()` method, which returns a `ListWrapper` around a specified array of objects.

Any changes made to the `List` are also made to the underlying array. This is a powerful method that allows any array of objects to be manipulated in any of the ways a `List` can be manipulated. It provides a link between arrays and the Java collections framework.

The various `sort()` methods sort an array (or a specified portion of an array) in place. Variants of the method are defined for arrays of each primitive type and for arrays of `Object`. For arrays of primitive types, the sorting is done according to the natural ordering of the type.

For arrays of objects, the sorting is done according to the specified `Comparator`, or, if the array contains only `java.lang.Comparable` objects, according to the ordering defined by that interface. When sorting an array of objects, a stable sorting algorithm is used so that the relative ordering of equal objects is not disturbed. (This allows repeated sorts to order objects by key and subkey, for example.)

The `binarySearch()` methods perform an efficient search (in logarithmic time) of a sorted array for a specified value. If a match is found in the array, `binarySearch()` returns the index of the match. If no match is found, the method returns a negative number.

For a negative return value `r`, the index `-(r+1)` specifies the array index at which the specified value can be inserted to maintain the sorted order of the array. When the array to be searched is an array of objects, the elements of the array must all implement `java.lang.Comparable`, or you must provide a `Comparator` object to compare them.

The equals() methods test whether two arrays are equal. Two arrays of primitive type are equal if they contain the same number of elements and if corresponding pairs of elements are equal according to the == operator.

Two arrays of objects are equal if they contain the same number of elements and if corresponding pairs of elements are equal according to the equals() method defined by those objects. The fill() methods fill an array or a specified range of an array with the specified value.

```
public class Arrays {  
    // No Constructor  
    // Public Class Methods  
    public static java.util.List asList (Object[ ] a);  
    public static int binarySearch (short[ ] a, short key);  
    public static int binarySearch (Object[ ] a, Object key);  
    public static int binarySearch (long[ ] a, long key);  
    public static int binarySearch (int[ ] a, int key);  
    public static int binarySearch (double[ ] a, double key);  
    public static int binarySearch (byte[ ] a, byte key);  
    public static int binarySearch (char[ ] a, char key);  
    public static int binarySearch (float[ ] a, float key);  
    public static int binarySearch (Object[ ] a, Object key, Comparator c);  
    public static boolean equals (boolean[ ] a, boolean[ ] a2);  
    public static boolean equals (byte[ ] a, byte[ ] a2);  
    public static boolean equals (float[ ] a, float[ ] a2);  
    public static boolean equals (double[ ] a, double[ ] a2);  
    public static boolean equals (int[ ] a, int[ ] a2);  
    public static boolean equals (long[ ] a, long[ ] a2);  
    public static boolean equals (char[ ] a, char[ ] a2);  
    public static boolean equals (short[ ] a, short[ ] a2);  
    public static boolean equals (Object[ ] a, Object[ ] a2);  
    public static void fill (double[ ] a, double val);  
    public static void fill (char[ ] a, char val);  
    public static void fill (short[ ] a, short val);  
    public static void fill (Object[ ] a, Object val);  
    public static void fill (float[ ] a, float val);  
    public static void fill (byte[ ] a, byte val);  
    public static void fill (int[ ] a, int val);  
    public static void fill (long[ ] a, long val);  
    public static void fill (boolean[ ] a, boolean val);  
    public static void fill (Object[ ] a, int fromIndex, int toIndex, Object val);  
    public static void fill (boolean[ ] a, int fromIndex, int toIndex, boolean val);  
    public static void fill (byte[ ] a, int fromIndex, int toIndex, byte val);  
    public static void fill (float[ ] a, int fromIndex, int toIndex, float val);
```

```

public static void fill (short[ ] a, int fromIndex, int toIndex, short val);
public static void fill (int[ ] a, int fromIndex, int toIndex, int val);
public static void fill (long[ ] a, int fromIndex, int toIndex, long val);
public static void fill (double[ ] a, int fromIndex, int toIndex, double val);
public static void fill (char[ ] a, int fromIndex, int toIndex, char val);
public static void sort (char[ ] a);
public static void sort (short[ ] a);
public static void sort (int[ ] a);
public static void sort (byte[ ] a);
public static void sort (double[ ] a);
public static void sort (float[ ] a);
public static void sort (long[ ] a);
public static void sort (Object[ ] a);
public static void sort (Object[ ] a, Comparator c);
public static void sort (short[ ] a, int fromIndex, int toIndex);
public static void sort (Object[ ] a, int fromIndex, int toIndex);
public static void sort (byte[ ] a, int fromIndex, int toIndex);
public static void sort (char[ ] a, int fromIndex, int toIndex);
public static void sort (float[ ] a, int fromIndex, int toIndex);
public static void sort (double[ ] a, int fromIndex, int toIndex);
public static void sort (int[ ] a, int fromIndex, int toIndex);
public static void sort (long[ ] a, int fromIndex, int toIndex);
public static void sort (Object[ ] a, int fromIndex, int toIndex, Comparator c);
}

```

4. DESCRIBE ABOUT THE CALENDAR CLASS IN JAVA.UTIL PACKAGE

This abstract class defines methods that perform date and time arithmetic. It also includes methods that convert dates and times to and from the machine-usable millisecond format used by the Date class and units such as minutes, hours, days, weeks, months, and years that are more useful to humans. As an abstract class, Calendar cannot be directly instantiated.

Instead, it provides static getInstance() methods that return instances of a Calendar subclass suitable for use in a specified or default locale with a specified or default time zone. See also Date, DateFormat, and TimeZone.

Calendar defines a number of useful constants. Some of these are values that represent days of the week and months of the year. Other constants, such as HOUR and DAY_OF_WEEK, represent various fields of date and time information. These field constants are passed to a number of Calendar methods, such as get() and set(), in order to indicate what particular date or time field is desired.

setTime() and the various set() methods set the date represented by a Calendar object. The add() method adds (or subtracts) values to a calendar field, incrementing

the next larger field when the field being set rolls over. roll() does the same, without modifying anything but the specified field. before() and after() compare two Calendar objects.

Many of the methods of the Calendar class are replacements for methods of Date that have been deprecated as of Java 1.1. While the Calendar class converts a time value to its various hour, day, month, and other fields, it is not intended to present those fields in a form suitable for display to the end user. That function is performed by the java.text.DateFormat class, which handles internationalization issues.

```
//  
protected Calendar ();  
protected Calendar (TimeZone zone, Locale aLocale);  
  
//  
public static final int AM ;  
public static final int AM_PM ;  
public static final int APRIL ;  
public static final int AUGUST ;  
public static final int DATE ;  
public static final int DAY_OF_MONTH ;  
public static final int DAY_OF_WEEK ;  
public static final int DAY_OF_WEEK_IN_MONTH ;  
public static final int DAY_OF_YEAR ;  
public static final int DECEMBER ;  
public static final int DST_OFFSET ;  
public static final int ERA ;  
public static final int FEBRUARY ;  
public static final int FIELD_COUNT ;  
public static final int FRIDAY ;  
public static final int HOUR ;  
public static final int HOUR_OF_DAY ;  
public static final int JANUARY ;  
public static final int JULY ;  
public static final int JUNE ;  
public static final int MARCH ;  
public static final int MAY ;  
public static final int MILLISECOND ;  
public static final int MINUTE ;  
public static final int MONDAY ;  
public static final int MONTH ;  
public static final int NOVEMBER ;  
public static final int OCTOBER ;  
public static final int PM ;  
public static final int SATURDAY ;  
public static final int SECOND ;
```

```

public static final int SEPTEMBER ;
public static final int SUNDAY ;
public static final int THURSDAY ;
public static final int TUESDAY ;
public static final int UNDECIMBER ;
public static final int WEDNESDAY ;
public static final int WEEK_OF_MONTH ;
public static final int WEEK_OF_YEAR ;
public static final int YEAR ;
public static final int ZONE_OFFSET ;

//
public static Locale[] getAvailableLocales ();
public static Calendar getInstance ();
public static Calendar getInstance (Locale aLocale);
public static Calendar getInstance (TimeZone zone);
public static Calendar getInstance (TimeZone zone, Locale aLocale);

//
public int getFirstDayOfWeek ();
public void setFirstDayOfWeek (int value);
public boolean isLenient ();
public void setLenient (boolean lenient);
public int getMinimalDaysInFirstWeek ();
public void setMinimalDaysInFirstWeek (int value);
public final java.util.Date getTime ();
public final void setTime (java.util.Date date);
public TimeZone getTimeZone ();
public void setTimeZone (TimeZone value);

//
public abstract void add (int field, int amount);
public boolean after (Object when);
public boolean before (Object when);
public final void clear ();
public final void clear (int field);
public final int get (int field);
1.2 public int getActualMaximum (int field);
1.2 public int getActualMinimum (int field);
public abstract int getGreatestMinimum (int field);
public abstract int getLeastMaximum (int field);
public abstract int getMaximum (int field);
public abstract int getMinimum (int field);
public final boolean isSet (int field);

```

```

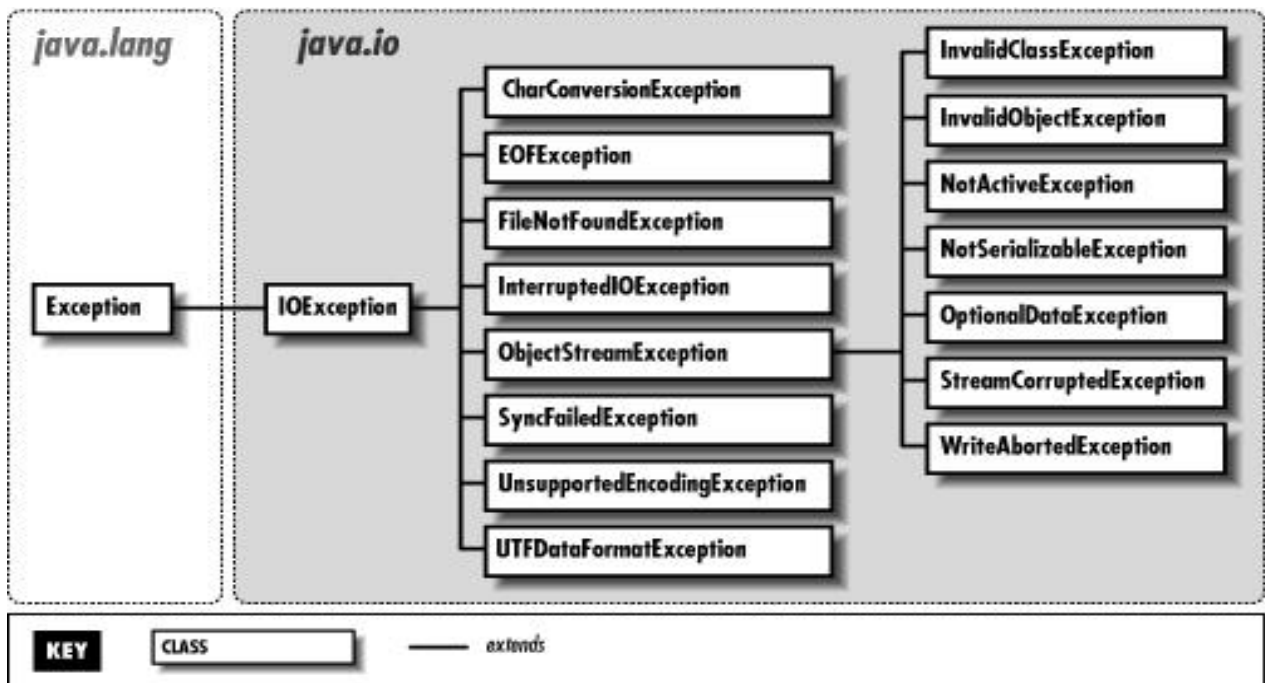
1.2 public void roll (int field, int amount);
    public abstract void roll (int field, boolean up);
    public final void set (int field, int value);
    public final void set (int year, int month, int date);
    public final void set (int year, int month, int date, int hour, int minute);
    public final void set (int year, int month, int date, int hour, int minute, int second);

//
    public Object clone ();
    public boolean equals (Object obj);
1.2 public int hashCode ();
    public String toString ();

//
    protected void complete ();
    protected abstract void computeFields ();
    protected abstract void computeTime ();
    protected long getTimeInMillis ();
    protected final int internalGet (int field);
    protected void setTimeInMillis (long millis);

//
    protected boolean areFieldsSet ;
    protected int[ ] fields ;
    protected boolean[ ] isSet ;
    protected boolean isTimeSet ;
    protected long time ;

```



5. EXPLAIN IN DETAIL ABOUT GENERICS

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects. Before generics, we can store any type of objects in collection i.e. non-generic.

Now generics, forces the java programmer to store specific type of objects.

Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

1) Type-safety : We can hold only a single type of objects in generics. It doesn't allow to store other objects.

2) Type casting is not required: There is no need to typecast the object.

Before Generics, we need to type cast.

```
List list = new ArrayList();  
list.add("hello");  
String s = (String) list.get(0); //typecasting
```

After Generics, we don't need to typecast the object.

```
List<String> list = new ArrayList<String>();  
list.add("hello");  
String s = list.get(0);
```

3) Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

```
List<String> list = new ArrayList<String>();  
list.add("hello");  
list.add(32); //Compile Time Error
```

Syntax to use generic collection

```
ClassOrInterface<Type>
```

Example to use Generics in java

```
ArrayList<String>
```

EXAMPLE OF GENERICS IN JAVA

Here, we are using the ArrayList class, but you can use any collection class such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, Comparator etc.


```

import java.util.*;

class TestGenerics1
{
    public static void main(String args[])
    {
        ArrayList<String> list=new ArrayList<String>();
        list.add("rahul");
        list.add("jai");
        //list.add(32);//compile time error

        String s=list.get(1);//type casting is not required
        System.out.println("element is: "+s);

        Iterator<String> itr=list.iterator();
        while(itr.hasNext())
        {
            System.out.println(itr.next());
        }
    }
}

```

Output:

```

element is: jai
rahul
jai

```

GENERIC CLASS

A class that can refer to any type is known as generic class. Here, we are using **T** type parameter to create the generic class of specific type.

Let's see the simple example to create and use the generic class.

Creating generic class:

```

class
MyGen<T>{ T obj;
void add(T obj){this.obj=obj;}
T get(){return obj;}
}

```

The T type indicates that it can refer to any type (like String, Integer, Employee etc.). The type you specify for the class, will be used to store and retrieve the data.

Using generic class:

Let's see the code to use the generic class.

```
class TestGenerics3{
    public static void main(String
    args[]){ MyGen<Integer> m=new
    MyGen<Integer>(); m.add(2);
    //m.add("vivek");//Compile time error
    System.out.println(m.get());
    }
}
```

Output: 2

GENERIC METHOD

Like generic class, we can create generic method that can accept any type of argument.

Let's see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

```
public class TestGenerics4{

    public static < E > void printArray(E[] elements) {
        for ( E element :
            elements){ System.out.println
            (element );
        }
        System.out.println();
    }
    public static void main( String args[] )
    { Integer[] intArray = { 10, 20, 30, 40,
    50 }; Character[] charArray = { 'J', 'A', 'V',
    'A' };

        System.out.println( "Printing Integer Array" );
        printArray( intArray );

        System.out.println( "Printing Character Array" );
        printArray( charArray );
    }
}
```

Output:

Printing Integer Array

30

40

50

Printing Character Array

J

A

V

A

6. WRITE IN DETAIL ABOUT INNER CLASSES IN JAVA

Java Inner Class

Java inner class or nested class is a class i.e. declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

Syntax of Inner class

```
class Java_Outer_class{
    //code
    class Java_Inner_class{
        //code
    }
}
```

Advantage of java inner classes

There are basically three advantages of inner classes in java. They are as follows:

1) Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.

2) Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.

3) **Code Optimization:** It requires less code to write.

TYPES OF NESTED CLASSES

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

1. Non-static nested class(inner class)
 - o a)Member inner class

- b) Anonymous inner class
 - c) Local inner class
2. Static nested class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

JAVA MEMBER INNER CLASS

A non-static class that is created inside a class but outside a method is called member inner class.

Syntax:

```
class Outer{
    //code
    class Inner{
        //code
    }
}
```

Java Member inner class example

In this example, we are creating msg() method in member inner class that is accessing the private data member of outer class.

```
class TestMemberOuter1{
    private int data=30;
    class Inner{
        void msg(){System.out.println("data is "+data);}
    }
    public static void main(String
    args[]){ TestMemberOuter1 obj=new
    TestMemberOuter1(); TestMemberOuter1.Inner
    in=obj.new Inner(); in.msg();
}
```

```
}
```

Output:

```
data is 30
```

Java Anonymous inner class

A class that have no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface. Java Anonymous inner class can be created by two ways:

1. Class (may be abstract or concrete).
2. Interface

Java anonymous inner class example using class

```
abstract class Person{  
    abstract void eat();  
}  
class TestAnonymousInner{  
    public static void main(String  
    args[]){ Person p=new Person(){  
        void eat(){System.out.println("nice fruits");}  
    };  
    p.eat();  
}  
}
```

Output:

```
nice fruits
```

Java anonymous inner class example using interface

```
interface Eatable{  
    void eat();  
}  
class TestAnonymousInner1{  
    public static void main(String  
    args[]){ Eatable e=new Eatable(){  
        public void eat(){System.out.println("nice fruits");}  
    };  
    e.eat();  
}  
}
```

Output:

```
nice fruits
```

JAVA LOCAL INNER CLASS

A class i.e. created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Java local inner class example

```
public class localInner1{
    private int data=30;//instance variable
    void display(){
        class Local{
            void msg(){System.out.println(data);}
        }
        Local l=new Local();
        l.msg();
    }
    public static void main(String
    args[]){ localInner1 obj=new
    localInner1(); obj.display();
    }
}
```

Output:

```
30
```

9. EXPLAIN IN DETAIL ABOUT JDBC API IN JAVA

JDBC is a standard interface for connecting to relational databases from Java.
Java API for connecting programs written in Java to the data in relational databases
The JDBC classes and interfaces are in the *java.sql* package.
JDBC 1.22 is part of JDK 1.1; **JDBC 2.0** is part of Java 2

The standard defined by Sun Microsystems, allowing individual providers to implement and extend the standard with their own JDBC drivers.

TASKS OF JDBC:

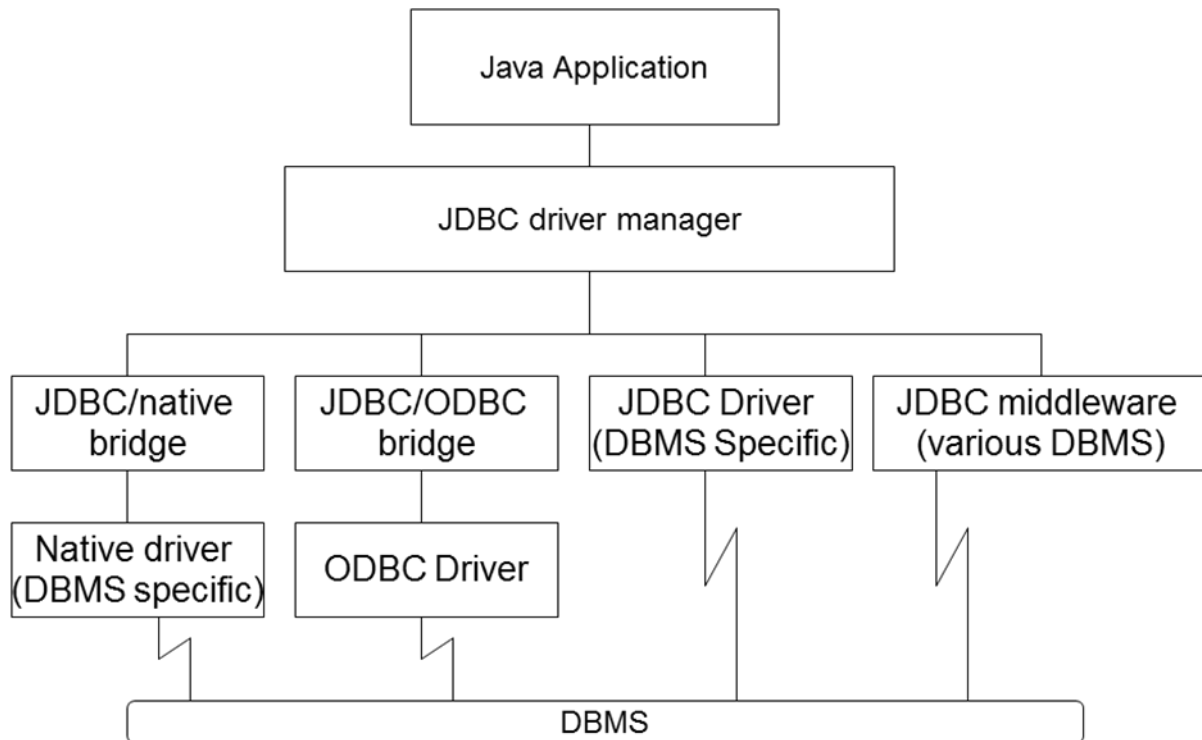
- 1) Establishes a connection with a database
- 2) Send SQL statements
- 3) Process the results

The JDBC API supports both two-tier and three-tier models for database access.

- > Two-tier model -- a Java applet or application interacts directly with the database.
- > Three-tier model -- introduces a middle-level server for execution of business logic:
 - > the middle tier to maintain control over data access.
 - > the user can employ an easy-to-use higher-level API which is translated by the middle tier into the appropriate low-level calls.

JDBC ARCHITECTURE:

JDBC Architecture



Class-I:

- JDBC:ODBC (mainly for Desktop Applications)
- > Use bridging technology
 - > Requires installation/configuration on client machines
 - > Not good for Web

Class-II:

- Native API Drivers (Vendor Specific drivers)
- > Requires installation/configuration on client machines
 - > Used to leverage existing CLI libraries
 - > Usually not thread-safe
 - > Mostly obsolete now
 - > e.g. Intersolv Oracle Driver, WebLogic drivers

Class-III:

- Network API
- > Calls middleware server, usually on database host
 - > Very flexible & allows access to multiple database using one driver

- > Only need to download one driver
- > But it's another server application to install and maintain
- > e.g. Symantec DBAnywhere

Class-IV:

Network Protocol Driver (used for Network based Applications)

- > Pure Java Drivers
- > Use Java networking libraries to talk directly to database engines
- > need to download a new driver for each database engine
- > e.g. Oracle, MySQL

OVERVIEW OF QUERYING A DATABASE WITH JDBC

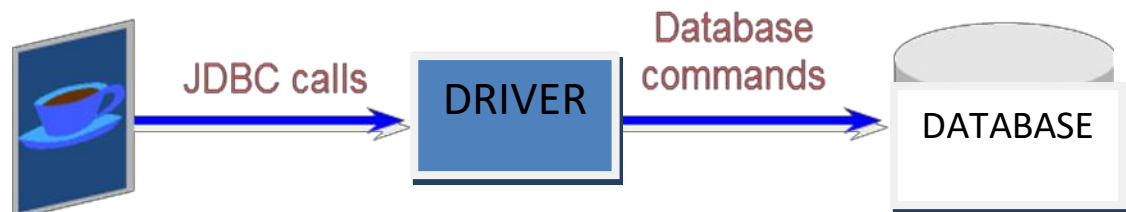
- 1) Connect
- 2) Query
- 3) Process Results
- 4) Close

1) CONNECT

- i) Register the driver
- ii) Connect to the Database

- i) Register the driver

A JDBC Driver is an interpreter that translates JDBC method calls to vendor-specific database commands
 Implements interfaces in java.sql
 Can also provide a vendor's extensions to the JDBC standard



Two Ways:

- 1) Using `Class.forName()`

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

- 2) Using `DriverManager.registerDriver()`

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
```

- ii) Connect to the Database

```
Connection conn = DriverManager.getConnection(URL, userid, password);
```

URL:

JDBC uses a URL to identify the database connection.

Jdbc:<subprotocol>:<subname>
Where
Subprotocol – Database Server name
Subname – Database Identifier

Ex:

```
Connection conn = DriverManager.getConnection  
("jdbc:oracle:thin:@myhost:1521:orcl",  
"scott", "tiger");
```

2) QUERY

- i) Create a Statement
- ii) Query The Database

- i) Create a Statement

The Statement Object

A Statement object sends your SQL statement to the database.
You need an active connection to create a JDBC statement.

Statement has three methods to execute a SQL statement:

- executeQuery() for QUERY statements
- executeUpdate() for INSERT, UPDATE, DELETE, or DDL statements
- execute() for either type of statement

- a. Create an empty statement object.

```
Statement stmt = conn.createStatement();
```

- b. Execute the statement.

```
ResultSet rset = stmt.executeQuery(statement);  
int count = stmt.executeUpdate(statement);  
boolean isquery = stmt.execute(statement);
```

The PreparedStatement Object

A PreparedStatement object holds precompiled SQL statements.

Use this object for statements you want to execute more than once.

A prepared statement can contain variables that you supply each time you execute the statement.

- a. Create an empty preparedstatement object.

Create the prepared statement, identifying variables with a question mark (?).

```
PreparedStatement pstmt = conn.prepareStatement("update  
ACME_RENTALS set STATUS = ? where RENTAL_ID  
= ?");
```

- b. Executing a Prepared Statement

1. Supply values for the variables

```
pstmt.setXXX(index, value);
```

2. Execute the statement

```
pstmt.executeQuery();  
pstmt.executeUpdate();
```

Eg:

```
PreparedStatement pstmt =  
conn.prepareStatement("update ACME_RENTALS  
set STATUS = ? where RENTAL_ID = ?");  
pstmt.setString(1, "OUT");  
pstmt.setInt(2, rentalid);  
pstmt.executeUpdate();
```

3. PROCESS RESULTS

- i) Step through the results
- ii) Assign Values to the results

The ResultSet Object

JDBC returns the results of a query in a ResultSet object.
A ResultSet maintains a cursor pointing to its current row of data.
Use next() to step through the result set row by row.
getString(), getInt(), and so on assign each value to a Java variable.

- i) Step through the result set:
while (rset.next())
{
...
}
- ii) Use getXXX() to get each column value

- a) Using Column Name

```
String val = rset.getString(colname);
```

- b) Using Column Index

```
String val = rset.getString(colIndex);
```

EG:

```
while (rset.next())  
{  
String title = rset.getString("TITLE");  
String year = rset.getString("YEAR");  
... // Process or display the data  
}
```

4. CLOSE

- i) Close the ResultSet object.
rset.close();
- ii) Close the Statement object.
stmt.close();
- iii) Close the connection (not necessary for server-side driver).
conn.close();

EXAMPLE PROGRAM:

```
public void AddStudent()
{
    try
    {
        Class.forName("com.mysql.jdbc.Driver"); Connection con =
        DriverManager.getConnection("jdbc:mysql://localhost:3306/sample","root","admin");
        int rollno = Integer.parseInt(txtRollno.getText());
        String name = txtName.getText();
        String department = txtDept.getText();
        int mark1 = Integer.parseInt(txtMark1.getText());
        int mark2 = Integer.parseInt(txtMark2.getText());
        int mark3 = Integer.parseInt(txtMark3.getText());
        int total = mark1 + mark2 + mark3;
        String result;
        if(mark1 >=50 && mark2 >=50 && mark3 >=50)
        {
            result = "Pass";
        }
        else
        {
            result = "Fail";
        }

        PreparedStatement ps=con.prepareStatement("INSERT INTO student
        (rollno,name,department,mark1,mark2,mark3,total,result) VALUES (?,?,?,?,?,?,?)");
        ps.setInt(1,rollno);
        ps.setString(2,name);
        ps.setString(3,department);
        ps.setInt(4,mark1);
        ps.setInt(5,mark2);
        ps.setInt(6,mark3);
        ps.setInt(7,total);
        ps.setString(8,result);

        ps.executeUpdate();

        ps.close();
        con.close();
    }
    catch (Exception ex)
```

```
{
    JOptionPane.showMessageDialog(this, ex);
}
}
```

10. EXPLAIN IN DETAIL ABOUT JAVA SECURITY

The Java platform provides a number of features designed to improve the **security** of Java applications. This includes enforcing runtime constraints through the use of the Java Virtual Machine (JVM), a security manager that sandboxes untrusted code from the rest of the operating system, and a suite of security APIs that Java developers can utilize.

Despite this, criticism has been directed at the programming language, and Oracle, due to an increase in malicious programs that revealed security vulnerabilities in the JVM, which were subsequently not properly addressed by Oracle in a timely manner.

Java's security model is focused on protecting users from hostile programs downloaded from untrusted sources across a network. To accomplish this goal, Java provides a customizable "sandbox" in which Java programs run.

A Java program must play only inside its sandbox. It can do anything within the boundaries of its sandbox, but it can't take any action outside those boundaries. The sandbox for untrusted Java applets, for example, prohibits many activities, including:

- Reading or writing to the local disk
- Making a network connection to any host, except the host from which the applet came
- Creating a new process
- Loading a new dynamic library and directly calling a native method

SAND BOX SECURITY MODEL

The sandbox security model is an intrinsic part of Java's architecture. The sandbox, a shell that surrounds a running Java program, protects the host system from malicious code. This security model helps give users confidence in downloading untrusted code across network.

The sandbox is designed into the Java virtual machine and Java API. It touches all corners of the architecture, but can be divided into four main components:

- Safety features (covered in this article)
- Class loaders (this will be covered next month)
- Class verification (this will be covered in the October issue)
- The security manager (this will be covered in the November issue)

Safety features built into the JVM

Several built-in security mechanisms are operating as Java virtual machine bytecodes. You have likely heard these mechanisms listed as features of the Java programming language that make Java programs robust. They are, not surprisingly, also features of the Java virtual machine. The mechanisms are:

- Type-safe reference casting
- Structured memory access (no pointer arithmetic)
- Automatic garbage collection (can't explicitly free allocated memory)
- Array bounds checking
- Checking references for null

Whenever you use an object reference, the JVM watches over you. If you attempt to cast a reference to a different type, the JVM makes sure the cast is valid. If you access an array, the JVM ensures the element you are requesting actually exists within the bounds of the array. If you ever try and use a null reference, the JVM throws an exception.

The following sections explain the basic concepts necessary to understand how this model works:

- Permissions
- Protection Domains and Security Policies
- Security Managers and Access Controllers

PERMISSIONS

A *permission* is a set of permissible operations on some set of resources. Every Java class loaded into a running environment is assigned a set of permissions according to some criteria, each permission granting a specific access to a particular resource. For example, a permission can constrain the access to a database or disallow the editing of a file.

In code-based security, permissions are granted based on code characteristics, such as where the code is coming from and whether it is digitally signed (and by whom). A *codebase* is a URL indicating code location, such as the following:

- file:** (any file on the local file system)
- http://*.oracle.com** (any file on any host at oracle.com)
- file:\${2ee.home}/lib/oc4j-internal.jar**

A *codesource* expands this concept to optionally include an array of certificates (stored in a Java keystore) to verify signed code originating from the location. A codesource is represented by a **java.security.CodeSource** instance, which is constructed by specifying a **java.net.URL** instance and an array of **java.security.cert.Certificate** instances.

The abstract Java class **java.security.Permission** represents a permission; concrete types, all derived from this class, include the following:

- java.security.AllPermission**
- java.lang.RuntimePermission** (includes only a resource target)
- java.io.FilePermission** (includes a resource and actions)

PROTECTION DOMAINS AND SECURITY POLICIES

A *protection domain* associates permissions with codesources. The policy currently in effect is what determines protection domains. A protection domain contains one or more codesources. It may also contain a Principal array describing who is executing the code, a classloader reference, and a permission set (**java.security.PermissionCollection** instance) representing a collection of Permission objects.

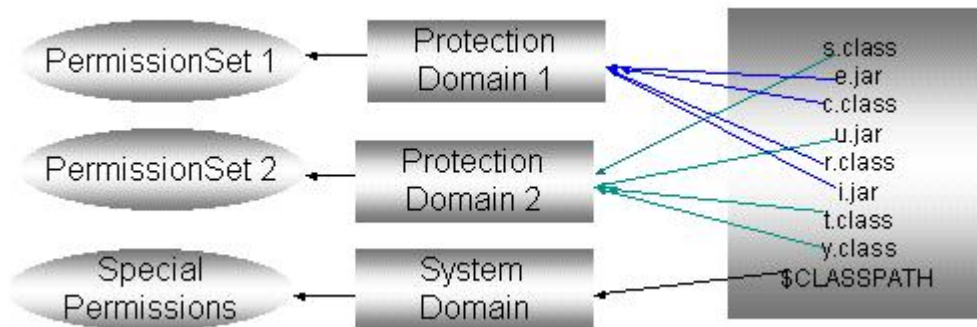
A *security policy* defines the protection domains of an environment, that is, it identifies the permissions assigned to classes from specified sources. The permissions assigned to a class by a protection domain are bound statically, when the class is loaded, or dynamically, when the executing code attempts a security-sensitive operation. Protection domains are specified in one or several policy files.

The following sample entry in a policy file illustrates how code located in the directory `/home/sysadmin` and signed by `jJoe` is granted read access to the file `/tmp/abc`:

```
grant signedby jJoe codeBase "file:/home/sysadmin/" {
    permission java.io.FilePermission "/tmp/abc", "read";
};
```

The **signedBy** clause in the preceding example is optional. If omitted, the permission is granted to all code in the specified location. [Figure 1-1](#) illustrates the relationship between classes, protection domains, and permission sets.

Figure 1-1 Associating classes with permissions through protection domains



SECURITY MANAGERS AND ACCESS CONTROLLERS

A *security manager* is the component of the Java security model that enforces the permissions granted to applications by security policies. For any security-sensitive operation that an application attempts, the security manager checks the application permissions and determines whether the operation should be allowed. The Java class **java.lang.SecurityManager** represents a security manager and includes several check methods to determine whether an operation should be allowed or a given permission is in effect.

An *access controller* is the object used by the security manager (or directly by an application, if the security manager is not enabled) to control operations and decisions. More specifically, an access controller:

- Decides whether access to a system resource should be allowed or denied, based on the current security policy in effect.

- Marks code as being privileged, thus affecting subsequent access determinations.

- Allows saving the current calling context so access-control decisions that consider the saved context can be made from other, different contexts

UNIT 5
2 MARKS

1. DEFINE JAVA BEAN

A "JAVA Bean" is a reusable software component that can be manipulated visually in a builder tool

2. STATE ANY TWO FEATURES OF JAVA BEANS

Support for "introspection" so that a builder tool can analyze how a bean works.

Support for "customization" to allow the customization of the appearance and behavior of a bean

3. WHAT ARE THE ADVANTAGES OF JAVA BEANS

Beans is platform independent, that means it can be run anywhere.

It can be run in any locale and is distributed in nature.

Methods, properties and events of Beans can be controlled

4. STATE THE RULES TO DEFINE JAVA BEANS

A JavaBean should:



be public



implement the Serializable interface



have a no-arg constructor

be derived from javax.swing.JComponent or java.awt.Component if it is visual

5. WHAT ARE THE PHASES IN JAVA BEAN CREATION

The Java Bean components can exist in one of the following three phases of development:

- Construction phase
- Build phase
- Execution phase

6. WHAT ARE THE ELEMENTS OF JAVA BEAN

Properties

Methods

Events

7. STATE SOME OF THE JAVA BEANS COMPONENT SPECIFICATION

- Customization
- Persistence
- Communication
- Introspection

8. WHAT IS APPLICATION BUILDER TOOL

Application Buidler Tool is used to configure and connect beans, and to create applications.

9. STATE ANY TWO PROPERTIES OF APPLICATION BUILDER TOOL

Palette is used to specify all the available beans. New beans can be added to palette anytime.

Worksheet is used to display beans in a GUI (Graphical User Interface). Beans can be dragged and dropped from palette to worksheet

10.NAME SOME APPLICATION BUILDER TOOLS

Java Workshop2.0

Jbuilder

Beans Development Kit

11. DEFINE BEAN DEVELOPMENT KIT

Provides a GUI to create, configure, and test JavaBeans.

Enables you to modify JavaBean properties and link multiple JavaBeans in an application using BDK

12. WHAT ARE THE USES OF BDK?

Provides a set of sample JavaBeans.

Enables you to associate pre-defined events with sample JavaBeans

13. DEFINE JAR FILE

JAR file allows you to efficiently deploy a set of classes and their associated resources. JAR file makes it much easier to deliver, install, and download. It is compressed

14. DEFINE INTROSPECTION

Introspection can be defined as the technique of obtaining information about bean properties, events and methods

Introspection is the automatic process by which a builder tool finds out which properties, methods, and events a bean supports

15. WHAT ARE THE WAYS TO IMPLEMENT INTROSPECTION**There are two ways**

With the first method, simple naming conventions are used. These allow the introspection mechanisms to infer information about a Bean.

In the second way, an additional class is provided that explicitly supplies this information

16. NAME THE INTERFACES IN JAVA BEANS PACKAGE

BeanInfo

AppletInitializer

Customizer

Visibility

17. DEFINE BEANINFO INTERFACE

Expose a Bean's features explicitly in a separate, associated class that implements the BeanInfo interface.

Provide a more descriptive display name, or additional information about a Bean feature

18. DEFINE PERSISTENCE AND STATE ITS TYPES

Persistence means an ability to save properties and events of our beans to non-volatile storage and retrieve later. It has the ability to save a bean to storage and retrieve it at a later time Configuration settings are saved It is implemented by Java serialization

Java Beans supports two forms of persistence:

Automatic persistence

External persistence

19. DEFINE CUSTOMIZATION IN JAVA BEANS

Customization is the ability of JavaBean to allow its properties to be changed in build and execution phase.

20. WHAT ARE THE SERVICES OF JAVA BEAN COMPONENT

Builder support
Layout
Interface publishing
Event handling
Persistence

21. DEFINE NETWORK

A **network** is a group of two or more computer systems linked together

22. STATE THE TYPES OF NETWORK

There are three types of network
LAN(Local Area Network)
MAN(Metropolitan Area Network)
WAN(Wide Area Network)

23. DEFINE PORT NUMBER

A **port number** is a way to identify a specific process to which an Internet or other network message is to be forwarded when it arrives at a server. The port numbers are divided into three ranges: the *well-known ports*, the *registered ports*, and the *dynamic or private ports*

24. DEFINE INETADDRESS CLASS

The java InetAddress or java.net.**InetAddress** class represents an IP address. The Java **InetAddress** class provides methods to get the IP of any host name

25. NAME THE METHODS IN INETADDRES

getByName(String host)- Determines the IP address of a host, given the host's name.
getHostAddress() - Returns the IP address string in textual presentation.
getHostName() - Gets the host name for this IP address.
getLocalHost()- Returns the local host

26. DEFINE SOCKET

A network **socket** is an endpoint of an inter-process communication across a computer network. Today, most communication between computers is based on the Internet Protocol; therefore most network **sockets** are Internet **sockets**

27. HOW JAVA IMPLEMENTS NETWORK PROGRAMMING?

- The term *network programming* refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.
- The java.net package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing you to write programs that focus on solving the problem at hand

28. DEFINE SERVER SOCKET AND SOME METHODS IN SERVER SOCKET

The **java.net.ServerSocket** class is used by server applications to obtain a port and listen for client requests

Here are some of the common methods of the ServerSocket class:

1. **getLocalPort()**- Returns the port that the server socket is listening on. This method is useful if you passed in 0 as the port number in a constructor and let the server find a port for you.
2. **accept()**- Waits for an incoming client. This method blocks until either a client connects to the server on the specified port or the socket times out, assuming that the time-out value has been set using the setSoTimeout() method. Otherwise, this method blocks indefinitely
3. **setSoTimeout(int timeout)**- Sets the time-out value for how long the server socket waits for a client during the accept().

29.DEFINE RMI

The Remote Method Invocation (RMI) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM

30.WHAT ARE THE STEPS TO WRITE RMI PROGRAM

There are 5 steps to follow

Server Side

1. Create the remote interface
2. Provide the implementation of the remote interface that extends UnicastRemoteObject
3. Create object for implementation class and register the object with the rmi registry

Client side

4. Create class in client side and get the remote reference by lookup the registry in the remote system
5. Using the remote reference, invoke the remote method

31.WHAT IS AN INTERNET ADDRESS?

Every computer connected to a network has a unique IP address. An IP address is a 32-bit number which has four numbers separated by periods. It is possible to connect to the Internet either directly or by using Internet Service Provider. By connecting directly to the Internet, the computer is assigned with a permanent IP address. In case connection is made using ISP, it assigns a temporary IP address for each session. A simple IP address is given below

80.0.0.78

32.WHAT ARE DATAGRAM?

Datagram is a type of packet that represents an entire communication. There is no necessity to have connection or disconnection stages when communicating using datagram. This is less reliable than communication using TCP/IP.

33.DEFINE PROXY SERVER.

A proxy server speaks the client side of a protocol to another server. This is often required when clients have certain restrictions on which servers they can connect to. Thus, a client would connect to a proxy server, which did not have such restrictions, and the proxy server would in

turn communicate for the client. A proxy server has the additional ability to filter certain requests or cache the results of those requests for future use.

34. WHAT IS THE USE OF URL CLASS IN JAVA? NAME ANY TWO METHODS IN IT.

URL stands for uniform Resource Locator and it points to resource files on the Internet. The URL has four components – the **protocol, IP address** or the **hostname, port number** and **actual file path**

Methods

getPort() get the port number
getHost() get the host name specified in URL
getFile() get the file name

35. WHAT IS SKELETON AND STUB? WHAT IS THE PURPOSE OF THOSE?

Stub is a client side representation of the server, which takes care of communicating with the remote server. Skeleton is the server side representation. But that is no more in use...it is deprecated long before in JDK

UNIT - V
11 MARKS**1. WRITE SHORT NOTES ON JAVA BEANS**

Software components are self-contained software units developed according to the motto “Developed them once, run and reused them everywhere”. Or in other words, reusability is the main concern behind the component model.

A software component is a reusable object that can be plugged into any target software application. You can develop software components using various programming languages, such as C, C++, Java, and Visual Basic

JAVA BEANS- DEFINITION

A “JAVA Bean” is a reusable software component that can be manipulated visually in a builder tool

JAVA BEANS – INTRODUCTION

The term software component model describe how to create and use reusable software components to build an application

Builder tool is nothing but an application development tool which lets you both to create new beans or use existing beans to create an application.

To enrich the software systems by adopting component technology JAVA came up with the concept called Java Beans.

Java provides the facility of creating some user defined components by means of Bean programming.

We create simple components using java beans.

We can directly embed these beans into the software

ADVANTAGES OF JAVA BEANS

- 1) Beans is platform independent, that means it can be run anywhere.
- 2) It can be run in any locale and is distributed in nature.
- 3) Methods, properties and events of Beans can be controlled.
- 4) It is easy to configure Java beans.
- 5) A bean can both receive and create events.
- 6) Configuration settings of a bean can be stored persistently and can be retrieved any time.

What can we do/create by using JavaBean:

There is no restriction on the capability of a Bean.

It may perform a simple function, such as checking the spelling of a document, or a complex function, such as forecasting the performance of a stock portfolio. A Bean may be visible to an end user. One example of this is a button on a graphical user interface.

Software to generate a pie chart from a set of data points is an example of a Bean that can execute locally.

Bean that provides real-time price information from a stock or commodities exchange.

Application Builder Tool:

It is used to configure and connect beans, and to create applications.

Properties of Application Builder Tool:




- 1) Palette is used to specify all the available beans. New beans can be added to palette anytime.
- 2) Worksheet is used to display beans in a GUI (Graphical User Interface). Beans can be dragged and dropped from palette to worksheet.
- 3) Editors are used for configuring a bean.
- 4) Commands can be used to check status of a bean.
- 5) Beans can be connected to one another.
- 6) Beans after configuration and connection can be stored persistently and can be retrieved any time.

Some Examples of Application Builder tools:

TOOL	VENDOR	DESCRIPTION
Java Workshop2.0	Sun Microsystems., Inc.,	Complete IDE that support applet, application and bean development
Visual age for java	IBM	Bean Oriented visual development toolset.
Jbuilder	Borland Inc.	Suit of bean oriented java development tool
Beans Development Kit	SunMicroSystems., Inc.,	Supports only Beans development

JAVA BEANS BASIC RULES

A JavaBean should:

-  be public
-  implement the Serializable interface
-  have a no-arg constructor
- be derived from javax.swing.JComponent or java.awt.Component if it is visual

The classes and interfaces defined in the java.beans package enable you to create JavaBeans.

The Java Bean components can exist in one of the following three phases of development:

- Construction phase
- Build phase
- Execution phase

ELEMENTS OF A JAVABEAN:**Properties**

Similar to instance variables.

A bean property is a named attribute of a bean that can affect its behavior or appearance. Examples of bean properties include color, label, font, font size, and display size.

Methods

Same as normal Java methods.

Every property should have accessor (get) and mutator (set) method.

All Public methods can be identified by the introspection mechanism.

There is no specific naming standard for these methods.

Events

Similar to Swing/AWT event handling.

THE JAVABEAN COMPONENT SPECIFICATION:

Customization: Is the ability of JavaBean to allow its properties to be changed in build and execution phase.

Persistence: Is the ability of JavaBean to save its state to disk or storage device and restore the saved state when the JavaBean is reloaded.

Communication: Is the ability of JavaBean to notify change in its properties to other JavaBeans or the container.

Introspection:- Is the ability of a JavaBean to allow an external application to query the properties, methods, and events supported by it.

SERVICES OF JAVABEAN COMPONENTS

Builder support: Enables you to create and group multiple JavaBeans in an application.

Layout- Allows multiple JavaBeans to be arranged in a development environment.

Interface publishing: Enables multiple JavaBeans in an application to communicate with each other.

Event handling: Refers to firing and handling of events associated with a JavaBean.

Persistence: Enables you to save the last state of JavaBean

FEATURES OF A JAVABEAN

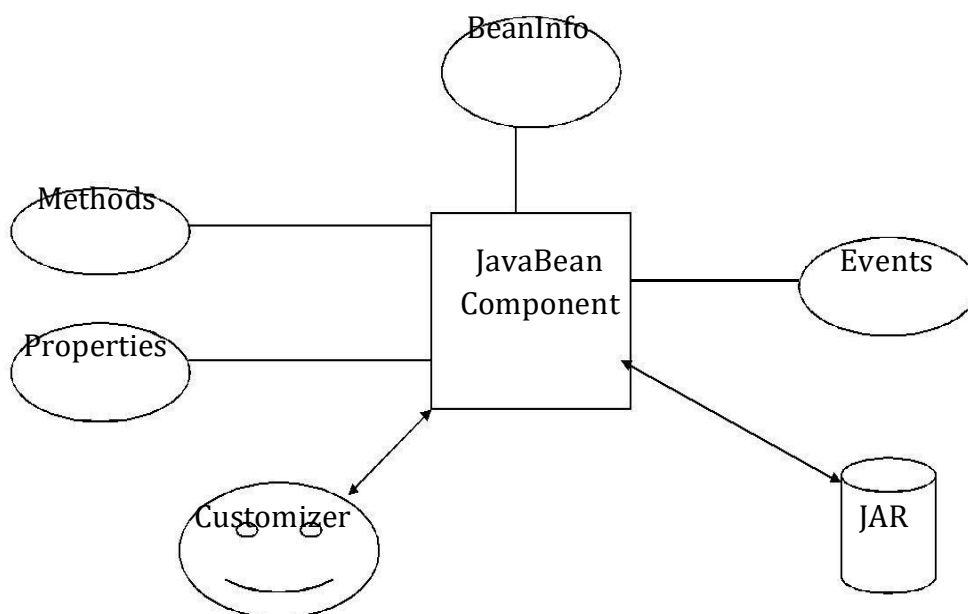
Support for “introspection” so that a builder tool can analyze how a bean works.

Support for “customization” to allow the customization of the appearance and behavior of a bean.

Support for “events” as a simple communication metaphor than can be used to connect up beans.

Support for “properties”, both for customization and for programmatic use.

Support for “persistence”, so that a bean can save and restore its customized state.



2. WRITE IN DETAIL ABOUT BEAN DEVELOPME KIT

Application Builder Tool:

It is used to configure and connect beans, and to create applications.

Properties of Application Builder Tool:

- 1) Palette is used to specify all the available beans. New beans can be added to palette anytime.
- 2) Worksheet is used to display beans in a GUI (Graphical User Interface). Beans can be dragged and dropped from palette to worksheet.
- 3) Editors are used for configuring a bean.
- 4) Commands can be used to check status of a bean.
- 5) Beans can be connected to one another.
- 6) Beans after configuration and connection can be stored persistently and can be retrieved any time.

Some Examples of Application Builder tools:

TOOL	VENDOR	DESCRIPTION
Java Workshop2.0	Sun Microsystems, Inc.,	Complete IDE that support applet, application and bean development
Visual age for java	IBM	Bean Oriented visual development toolset.
Jbuilder	Borland Inc.	Suit of bean oriented java development tool
Beans Development Kit	SunMicroSystems, Inc.,	Supports only Beans development

BEANS DEVELOPMENT KIT

Is a development environment to create, configure, and test JavaBeans.

The features of BDK environment are:

- Provides a GUI to create, configure, and test JavaBeans.
- Enables you to modify JavaBean properties and link multiple JavaBeans in an application using BDK.
- Provides a set of sample JavaBeans.
- Enables you to associate pre-defined events with sample JavaBeans.

Identifying BDK Components

- Execute the run.bat file of BDK to start the BDK development environment.
- The components of BDK development environment are:
 - ToolBox
 - BeanBox
 - Properties
 - Method Tracer

TOOLBOX WINDOW:

Lists the sample JavaBeans of BDK.

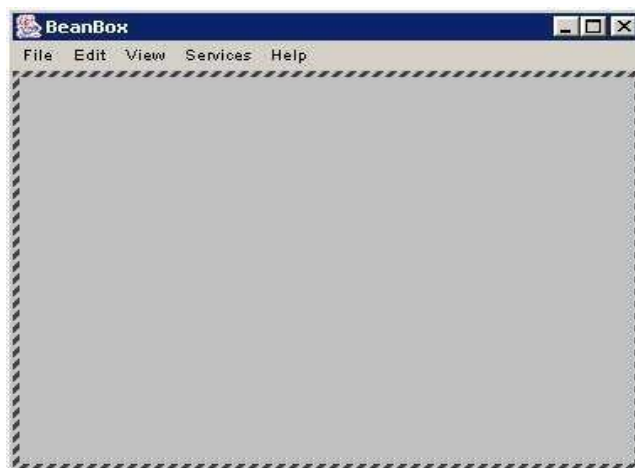
The following figure shows the ToolBox window:



BEANBOX WINDOW:

Is a workspace for creating the layout of JavaBean application.

The following figure shows the BeanBox window:



PROPERTIES WINDOW:

Displays all the exposed properties of a JavaBean. You can modify JavaBean properties in the properties window.

The following figure shows the Properties window:

**METHOD TRACER WINDOW:**

Displays the debugging messages and method calls for a JavaBean application.

The following figure shows the Method Tracer window:

**STEPS TO DEVELOP A USER-DEFINED JAVABEAN:**

1. Create a directory for the new bean
2. Create the java bean source file(s)
3. Compile the source file(s)
4. Create a manifest file
5. Generate a JAR file
6. Start BDK
7. Load Jar file
8. Test.

1. Create a directory for the new bean

Create a directory/folder like C:\Beans

2. Create bean source file - MyBean.java

```
import java.awt.*;
public class MyBean extends Canvas
{
    public MyBean()
    {
        setSize(70,50);
        setBackground(Color.green);
    }
}
```

3. Compile the source file(s)

C:\Beans >Javac MyBean.java

4. Create a manifest file

Manifest File

- The manifest file for a JavaBean application contains a list of all the class files that make up a JavaBean.
- The entry in the manifest file enables the target application to recognize the JavaBean classes for an application.
- For example, the entry for the MyBean JavaBean in the manifest file is as shown:

Manifest-Version: 1.0 Name: MyBean.class Java-Bean: true
--

Note: write that 2 lines code in the notepad and save that file as MyBean.mf

The rules to create a manifest file are:

- Press the Enter key after typing each line in the manifest file.
- Leave a space after the colon.
- Type a hyphen between Java and Bean.
- No blank line between the Name and the Java-Bean entry.

5. Generate a JAR file

Syntax for creating jar file using manifest file

```
C:\Beans >jar cfm MyBean.jar MyBean.mf MyBean.class
```

JAR file:

JAR file allows you to efficiently deploy a set of classes and their associated resources. JAR file makes it much easier to deliver, install, and download. It is compressed.

Java Archive File

- The files of a JavaBean application are compressed and grouped as JAR files to reduce the size and the download time of the files.
- The syntax to create a JAR file from the command prompt is:
jar <options> <file_names>
- The file_names is a list of files for a JavaBean application that are stored in the JAR file.

The various options that you can specify while creating a JAR file are:

A: Indicates the new JAR file is created.

B: Indicates that the first file in the file_names list is the name of the JAR file.

C: Indicates that the second file in the file_names list is the name of the manifest file.

D: files and resources in the JAR file are to be displayed in a tabular format.

E: Indicates that the JAR file should generate a verbose output.

F: Indicates that the files and resources of a JAR file are to be extracted. **o:** Indicates that the JAR file should not be compressed.

G: Indicates that the manifest file is not created.

6. Start BDk

Go to-> C:\bdk1_1\beans\beanbox

Click on run.bat file. When we click on run.bat file the BDk software automatically started.

7. Load Jar file

Go to

Beanbox->File->Load jar. Here we have to select our created jar file when we click on ok, our bean(userdefined) MyBean appear in the ToolBox.

8. Test our created user defined bean

Select the MyBean from the ToolBox when we select that bean one + simple appear then drag that Bean in to the Beanbox.

If you want to apply events for that bean, now we apply the events for that Bean.

3. WRITE IN DETAIL ABOUT INTROSPECTION, PROPERTIES AND PERSISTENCE IN JAVA BEANS

Introspection:

Introspection can be defined as the technique of obtaining information about bean properties, events and methods.

Basically introspection means analysis of bean capabilities.

Introspection is the automatic process by which a builder tool finds out which properties, methods, and events a bean supports.

Introspection describes how methods, properties, and events are discovered in the beans that you write.

This process controls the publishing and discovery of bean operations and properties

Without introspection, the JavaBeans technology could not operate.

4. DK Introspection:

Allows automatic analysis of a java beans component

Enables a builder tool to analyze how a bean works.

(Or)

A mechanism that allows classes to publish the operations and properties they support and a mechanism to support the discovery of such mechanism.

Introspection can be defined as the technique of obtaining information about bean properties, events and methods.

Basically introspection means analysis of bean capabilities

WAYS TO PERFORM INTROSPECTION

There are two ways

- 1) With the first method, simple naming conventions are used. These allow the introspection mechanisms to infer information about a Bean.
- 2) In the second way, an additional class is provided that explicitly supplies this information.

Design patterns for JavaBean Properties:-

A property is a subset of a Bean's state.

A bean *property* is a named attribute of a bean that can affect its behavior or appearance. Examples of bean properties include color, label, font, font size, and display size.

Properties are the private data members of the JavaBean classes.

Properties are used to accept input from an end user in order to customize a JavaBean

Properties can retrieve and specify the values of various attributes, which determine the behavior of a JavaBean.

Types of JavaBeans Properties

- Simple properties
- Boolean properties
- Indexed properties

Simple Properties:

Simple properties refer to the private variables of a JavaBean that can have only a single value. Simple properties are retrieved and specified using the get and set methods respectively.

A read/write property has both of these methods to access its values. The get method used to read the value of the property .The set method that sets the value of the property.

The setXXX() and getXXX() methods are the heart of the java beans properties mechanism. This is also called getters and setters. These accessor methods are used to set the property .

The syntax of get method is:

```
public return_type get<PropertyName>()
public T getN();
public void setN(T arg)
```

N is the name of the property and T is its type

Ex:

```
private int id;
public void setId(int id)
{
    this.id = id;
}

public String getName()
{
    return name;
}
```

Boolean Properties:

A Boolean property is a property which is used to represent the values True or False.

Have either of the two values, TRUE or FALSE.

It can identified by the following methods:

Syntax:

Let N be the name of the property and T be the type of the value then

```
public boolean isN();
public void setN(Boolean parameter);
public boolean getN();
public boolean is<PropertyName>()
```

```
public boolean get<PropertyName>()
```

First or second pattern can be used to retrieve the value of a Boolean.

- public void set<PropertyName>(boolean value)

For getting the values isN() and getN() methods are used and for setting the Boolean values setN() method is used.

Ex:

```
public Boolean dotted=false;
```

```
public boolean isDotted()
```

```
{
    return dotted;
}
```

```
public void setDotted(boolean dotted)
```

```
{
    this.dotted=dotted;
}
```

Indexed Properties:

Indexed Properties are consists of multiple values. If a simple property can hold an array of value they are no longer called simple but instead indexed properties. The method's signature has to be adapted accordingly.

An indexed property may expose set/get methods to read/write one element in the array (so-called 'index getter/setter') and/or so-called 'array getter/setter' which read/write the entire array.

Indexed Properties enable you to set or retrieve the values from an array of property values.

Indexed Properties are retrieved using the following get methods:

Syntax:

```
public int[] get<PropertyName>()
```

Ex:

```
private double data[ ];
public double getData(int index)
{
    return data[index];
}
public void setData(int index, double value)
{
    data[index] = value;
}
public double[ ] getData( )
{
    return data;
}
public void setData(double[ ] values)
{
    data = new double[values.length];
    System.arraycopy(values, 0, data, 0, values.length);
}
```


The properties window of BDK does not handle indexed properties. Hence the output cannot be displayed here.

Bound Properties:

A bean that has a bound property generates an event when the property is changed.

Bound Properties are the properties of a JavaBean that inform its listeners about changes in its values.

Bound Properties are implemented using the `PropertyChangeSupport` class and its methods.

Bound Properties are always registered with an external event listener.

The event is of type *PropertyChangeEvent* and is sent to objects that previously registered an interest in receiving such notifications bean with bound property - Event source Bean implementing listener -- event target

In order to provide this notification service a Java Bean needs to have the following two methods:

```
public void addPropertyChangeListener(PropertyChangeListener p)
{
    changes.addPropertyChangeListener(p);
}

public void removePropertyChangeListener(PropertyChangeListener p)
{
    changes.removePropertyChangeListener(p);
}
```

Constrained Properties:

It generates an event when an attempt is made to change its value. Constrained Properties are implemented using the *PropertyChangeEvent* class

The prototype of the get method is:

Syntax: `public String get<ConstrainedPropertyName>()`

Can be specified using the set method.

The prototype of the set method is:

Syntax : `public void set<ConstrainedPropertyName>(String str) throws PropertyVetoException`

DESIGN PATTERNS FOR EVENTS:

Handling Events in JavaBeans:

Enables Beans to communicate and connect together.

Beans generate events and these events can be sent to other objects.

Event means any activity that interrupts the current ongoing activity.

Example: mouse clicks, pressing key...

User-defined JavaBeans interact with the help of user-defined events, which are also called custom events. You can use the Java event delegation model to handle these custom events.

The components of the event delegation model are:

- **Event Source:** Generates the event and informs all the event listeners that are registered with it.
- **Event Listener:** Receives the notification, when an event source generates an event.
- **Event Object:** Represents the various types of events that can be generated by the event sources.

Creating Custom Events:

The classes and interfaces that you need to define to create the custom JavaBean events are:

- An event class to define a custom JavaBean event.
- An event listener interface for the custom JavaBean event.
- An event handler to process the custom JavaBean event.
- A target Java application that implements the custom event.

Creating the Event Class:

The event class that defines the custom event extends the EventObject class of the java.util package.

For example,

```
public class NumberEvent extends EventObject
{
    public int number1,number2;
    public NumberEvent(Object o,int number1,int number2)
    {
        super(o);
        this.number1=number1;
        this.number2=number2;
    }
}
```

Beans can generate events and send them together objects.

Creating Event Listeners

- When the event source triggers an event, it sends a notification to the event listener interface.
- The event listener interface implements the java.util.EventListener interface.

Syntax:

```
public void addTListener(TListener eventListener);
public void addTListener(TListener eventListener)throws TooManyListeners;
public void removeTListener(TListener eventListener);
```

The target application that uses the custom event implements the custom listener.

For example,

```
public interface NumberEnteredListener extends EventListener
{
    public void arithmeticPerformed(NumberEvent mec);
}
```

Creating Event Handler

Custom event handlers should define the following methods:

- addXXListener(): Registers listeners of a JavaBean event.
- fireXX(): Notifies the listeners of the occurrence of a JavaBean event.
- removeXXListener(): Removes a listener from the list of registered listeners of a JavaBean.

PERSISTENCE

Persistence means an ability to save properties and events of our beans to non-volatile storage and retrieve later. It has the ability to save a bean to storage and retrieve it at a later time Configuration settings are saved It is implemented by Java serialization.

If a bean inherits directly or indirectly from Component class it is automatically Serializable. Transient keyword can be used to designate data members of a Bean that should not be serialized.

Enables developers to customize Beans in an application builder, and then retrieve those Beans, with customized features intact, for future use, perhaps in another environment.

Java Beans supports two forms of persistence:

Automatic persistence
External persistence

Automatic Persistence:

Automatic persistence are java's built-in serialization mechanism to save and restore the state of a bean.

External Persistence:

External persistence, on the other hand, gives you the option of supplying your own custom classes to control precisely how a bean state is stored and retrieved.

4) WRITE IN DETAIL ABOUT BEANINFO INTERFACE IN JAVA BEANS**INTERFACES IN JAVA BEANS**

The JavaBeans functionality is provided by a set of classes and interfaces in the java.beans package.

Interface	Description
AppletInitializer	Methods in this interface are used to initialize Beans that are also applets.
BeanInfo	This interface allows the designer to specify information about the events, methods and properties of a Bean.
Customizer	This interface allows the designer to provide a graphical user interface through which a bean may be configured.
DesignMode	Methods in this interface determine if a bean is executing in design mode.
ExceptionListener	A method in this interface is invoked when an exception has occurred.
PropertyChangeListener	A method in this interface is invoked when a bound property is changed.
PropertyEditor	Objects that implement this interface allow the designer to change and display property values.
VetoableChangeListener	A method in this interface is invoked when a Constrained property is changed.
Visibility	Methods in this interface allow a bean to execute in environments where the GUI is not available.
USING BEANINFO INTERFACE	

you can *explicitly* expose a Bean's features in a separate, associated class that implements the BeanInfo interface.

By associating a BeanInfo class with your Bean, you can:

Expose only those features you want to expose.

Rely on BeanInfo to expose some Bean features while relying on low-level reflection to expose others.

Associate an icon with the target Bean.

Specify a customizer class.

Segregate features into normal and expert categories.

Provide a more descriptive display name, or additional information about a Bean feature.

BEANINFO INTERFACE

A bean implementor who wishes to provide explicit information about their bean may provide a BeanInfo class that implements this BeanInfo interface and provides explicit information about the methods, properties, events, etc, of their bean.

A bean implementor doesn't need to provide a complete set of explicit information. You can pick and choose which information you want to provide and the rest will be obtained by automatic analysis using low-level reflection of the bean classes' methods and applying standard design patterns.

You get the opportunity to provide lots and lots of different information as part of the various XYZDescriptor classes. But don't panic, you only really need to provide the minimal core information required by the various constructors.

METHODS IN BEANINFO INTERFACE

Method Summary

<u>BeanInfo[]</u>	<u>getAdditionalBeanInfo()</u> This method allows a BeanInfo object to return an arbitrary collection of other BeanInfo objects that provide additional information on the current bean.
<u>BeanDescriptor</u>	<u>getBeanDescriptor()</u> Gets the beans BeanDescriptor.
int	<u>getDefaultEventIndex()</u> A bean may have a "default" event that is the event that will mostly commonly be used by humans when using the bean.
int	<u>getDefaultPropertyIndex()</u> A bean may have a "default" property that is the property that will mostly commonly be initially chosen for update by human's who are

	customizing the bean.
<u>EventSetDescriptor[]</u>	<u>getEventSetDescriptors()</u> Gets the beans EventSetDescriptors.
<u>Image</u>	<u>getIcon(int iconKind)</u> This method returns an image object that can be used to represent the bean in toolboxes, toolbars, etc.
<u>MethodDescriptor[]</u>	<u>getMethodDescriptors()</u> Gets the beans MethodDescriptors.
<u>PropertyDescriptor[]</u>	<u>getPropertyDescriptors()</u> Gets the beans PropertyDescriptors.

Feature Descriptors

BeanInfo classes contain *descriptors* that precisely describe the target Bean's features.

The BDK implements the following descriptor classes:

FeatureDescriptor is the base class for the other descriptor classes. It declares the aspects common to all descriptor types.

BeanDescriptor describes the target Bean's class type and name, and describes the target Bean's customizer class if it exists.

PropertyDescriptor describes the target Bean's properties.

IndexedPropertyDescriptor is a subclass of PropertyDescriptor, and describes the target Bean's indexed properties.

EventSetDescriptor describes the events the target Bean fires.

MethodDescriptor describes the target Bean's methods.

ParameterDescriptor describes method parameters.

The BeanInfo interface declares methods that return arrays of the above descriptors.

Creating a BeanInfo Class

Here are the general steps to make a BeanInfo class:

1. **Name your BeanInfo class.** You must append the string "BeanInfo" to the target class name. If the target class name is ExplicitButton, then its associated Bean information class must be named ExplicitButtonBeanInfo
2. **Subclass SimpleBeanInfo.** This is a convenience class that implements BeanInfo methods to return null, or an equivalent no-op value.

```
public class ExplicitButtonBeanInfo extends SimpleBeanInfo {
```

Using `SimpleBeanInfo` saves you from implementing all the `BeanInfo` methods; you only have to override those methods you need.

3. **Override the appropriate methods to return the properties, methods, or events that you want exposed**

There are two important things to note here:

- If you leave a descriptor out, that property, event or method will *not* be exposed. In other words, you can selectively expose properties, events, or methods by leaving out those you don't want exposed.
- If a feature's getter (for example, `getMethodDescriptor()`) method returns null, low-level reflection is then used for that feature. This means, for example, that you can explicitly specify properties, and let low-level reflection discover the methods. If you don't override the `SimpleBeanInfo` default method, which returns null, low-level reflection will be used for that feature.

4. **Specify the target Bean class, and, if the Bean has a customizer, specify it also.**

```
public BeanDescriptor getBeanDescriptor() {
    return new BeanDescriptor(beanClass, customizerClass);
}
...
private final static Class beanClass = ExplicitButton.class;
private final static Class customizerClass = OurButtonCustomizer.class;
```

Keep the `BeanInfo` class in the same directory as its target class.

The `BeanBox` first searches for a target Bean's `BeanInfo` class in the target Bean's package path.

If no `BeanInfo` is found, then the Bean information package search path (maintained by the `Introspector`) is searched.

The default Bean information search path is `sun.beans.infos`.

If no `BeanInfo` class is found, then low-level reflection is used to discover a Bean's features.

Using BeanInfo to Control What Features are Exposed

By using a `BeanInfo` class, you can expose subsets of a particular Bean feature. For example, by not returning a method descriptor for a particular method, that method will not be exposed in a builder tool.

When you use a BeanInfo class:

Base class features will *not* be exposed. You can retrieve base class features by using the BeanInfo.getAdditionalBeanInfo method.

Properties, events, or methods that have no descriptor will *not* be exposed. For a particular feature, only those items returned in the descriptor array will be exposed. For example, if you return descriptors for all your Bean methods except foo, then foo will not be exposed.

Low-level reflection will be used for features with getter methods returning null. For example if your BeanInfo class contains this method implementation:

```
public MethodDescriptor[] getMethodDescriptors()
{
    return null;
}
```

Then low-level reflection will be used to discover your Bean's public methods.

Locating BeanInfo Classes

Before examining a Bean, the Introspector will attempt to find a BeanInfo class associated with the Bean.

By default, the Introspector takes the target Bean's fully qualified package name, and appends "BeanInfo" to form a new class name.

For example, if the target Bean is sunw.demo.buttons.ExplicitButton, then the Introspector will attempt to locate sunw.demo.buttons.ExplicitButtonBeanInfo.

If that fails, then each package in the BeanInfo search path is searched.

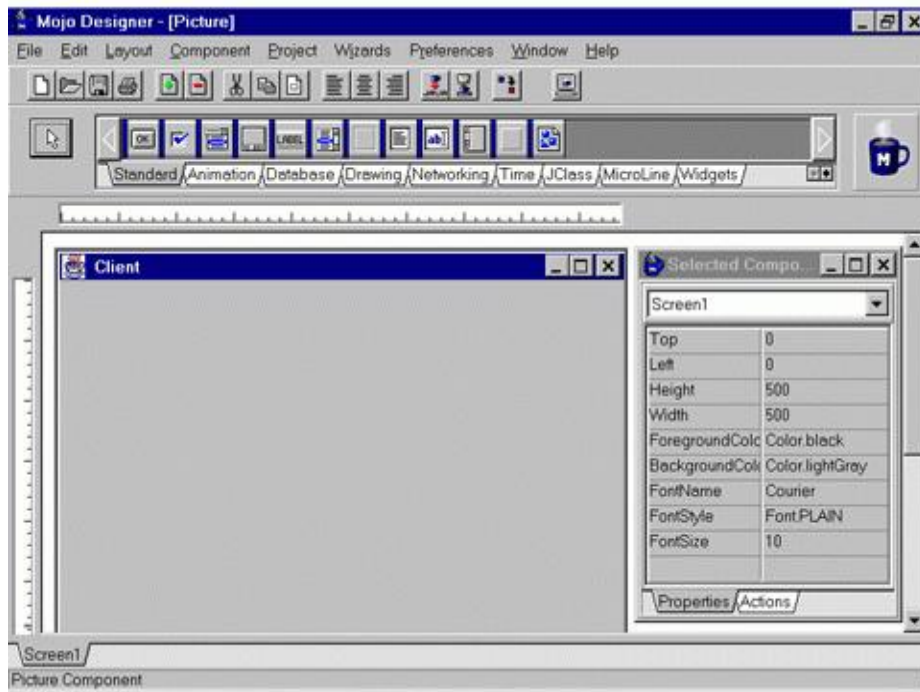
The BeanInfo search path is maintained by Introspector.setBeanInfoSearchPath() and Introspector.getBeanInfoSearchPath().

5. DESCRIBE IN DETAIL ABOUT JAVA BEANS API WITH BEAN BUILDER

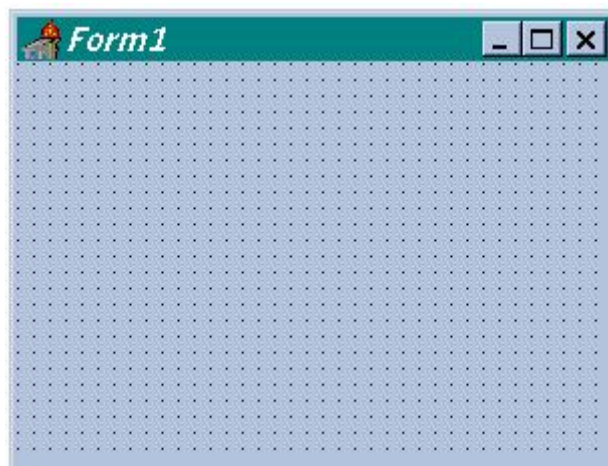
Application Builder Tools

The primary purpose of beans is to enable the visual construction of applications. You've probably used or seen applications like Visual Basic, Visual Age, or Delphi. These tools are referred to as visual ***application builders***, or ***builder tools*** for short.

Typically such tools are GUI applications, although they need not be. There is usually a palette of components available from which a program designer can drag items and place them on a *form* or *client window*.

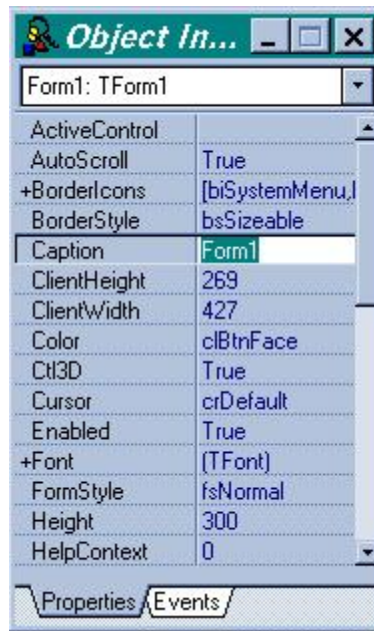


In Windows environments the form is often called the client window area.

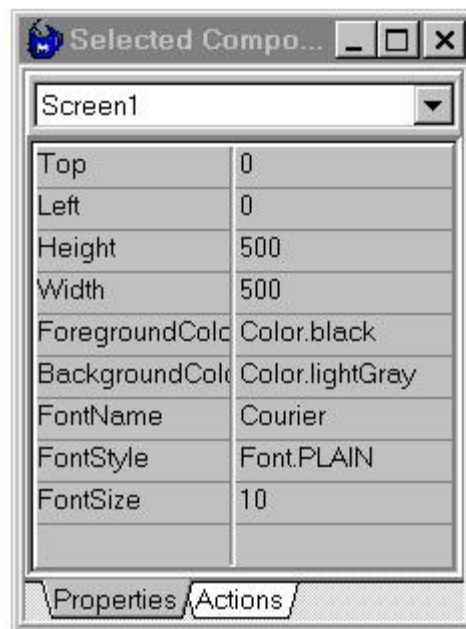


The form, or client window, represents the target application under construction and is presented, during design, as it will appear when it runs independently of the builder program.

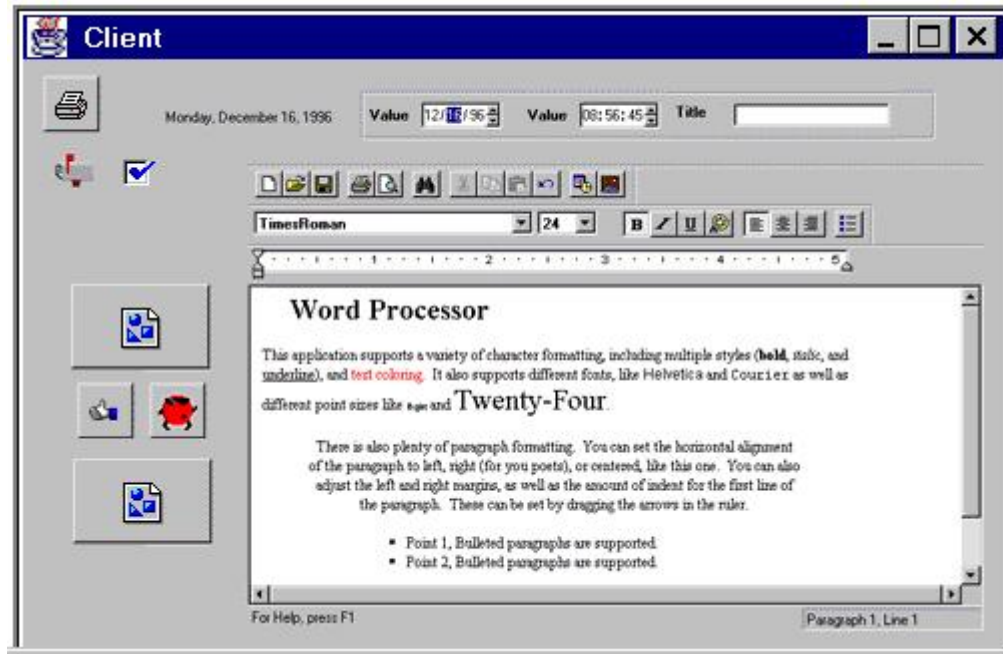
The most tell-tale window shared by many popular application builders is a *property sheet*, sometimes called a *property editor* or simply a *properties window*.



A property sheet is used to modify *properties* and *events* associated with components. In keeping with Java AWT terminology some property editors use the term *action* in place of event.



Applications built with powerful components can appear complex even when they take little effort to build.



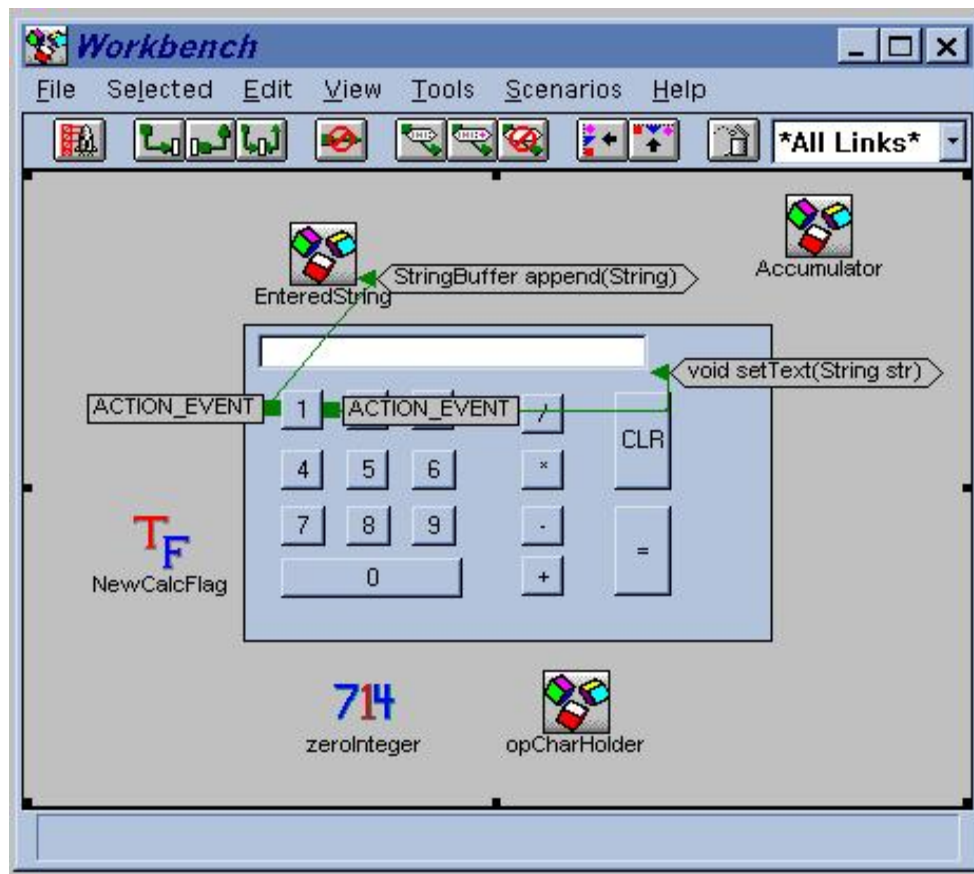
Linking Components in Builder Tools

The calculator program shown earlier is a good example of building a complex program with no handwritten code.



The calculator was built without writing any Java code. Instead, components were linked by using a mouse.

Generated events and event-handler methods were selected through pop-up menus.



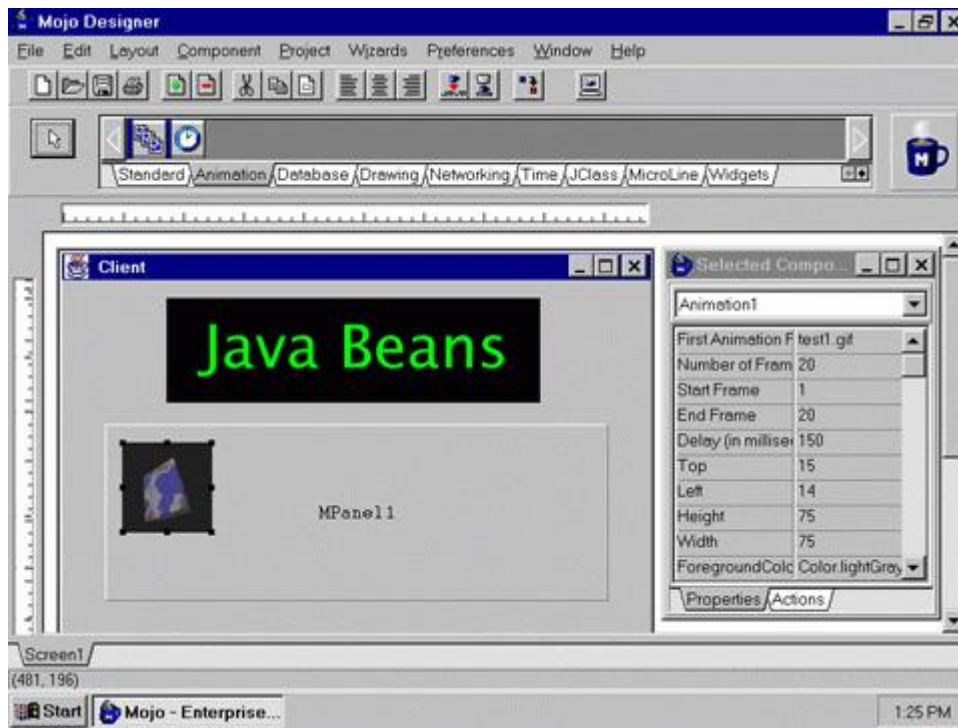
Every time a button is pressed, an event is fired and side effects result. Example side effects include appending digits to the invisible EnteredString storage buffer, performing a math operation, or displaying text in the display pane.

To program an event sequence, select a button as the event source, and indicate that you want to fire an event through a keystroke or menu command.

You'll see a menu, or a property sheet, listing events that can be fired. Select an event, then drag a link to the target object which should receive the event. You will be presented with a menu listing event handlers available for the target object.

The following sequence of screens shows a more complex example in a different builder tool. An animation component is added to a subpanel within a client window.

First an Animation Bean is selected from the builder's palette and dropped on a panel object in the client form.



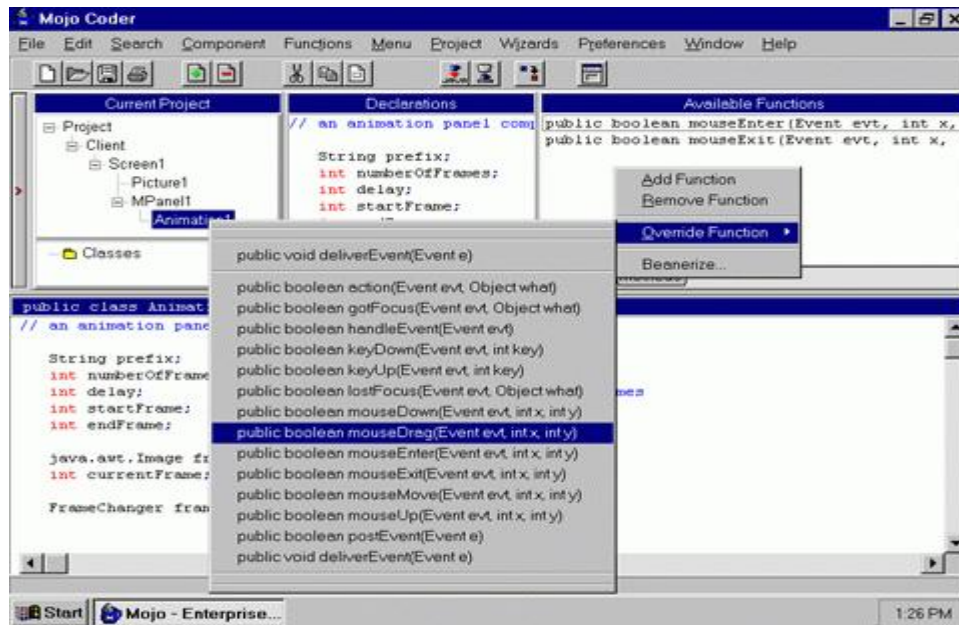
A Bean is customized by selecting action items from a property sheet, which in turn invokes a custom Bean editor to select a specific event from a list of events that can be generated by an Animation Bean.

A list of events is acquired by introspection; the builder tool uses Java introspection APIs to query beans dropped on the form.

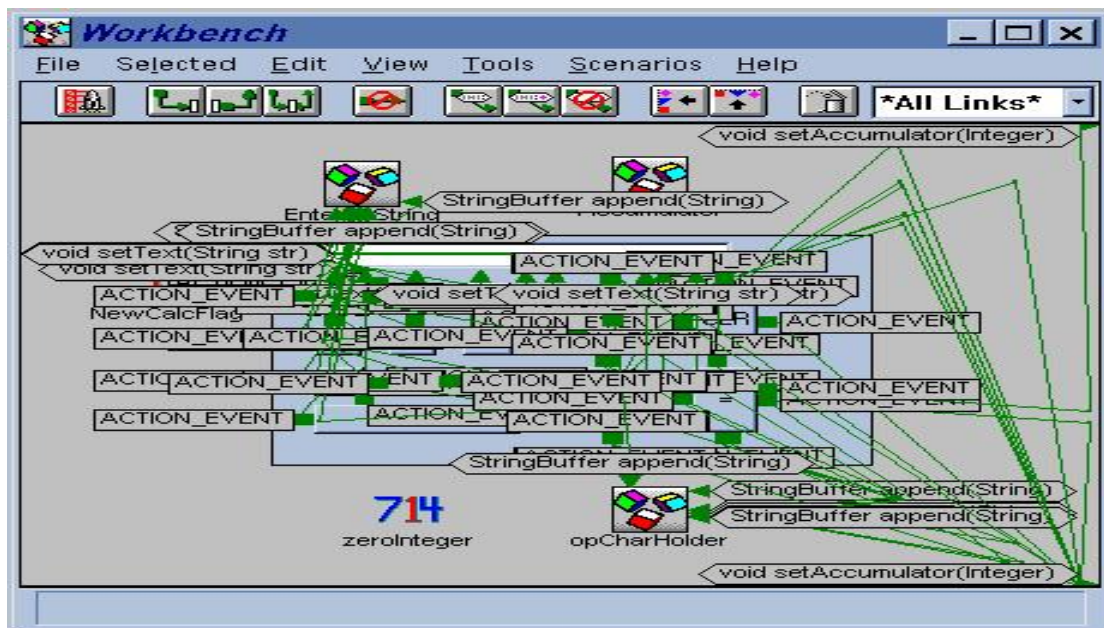


Default behavior for event handlers is usually acceptable, but in some cases you'll want to manually override handler definitions with a text editor.

In this builder tool, much of the programming, including selection of events and event handlers, is done using a class hierarchy browser. After selecting the event-handler method, a template for the event handler is generated, while you flesh out details manually by writing code in the browser.



However not all programs require text editing. As mentioned, the calculator example was built without resorting to text coding. The maze of links required to hookup components can be overwhelming, as you can see in the following figure.



But, most builder tools allow you to easily edit links and filter the display of links so that you see only the level of detail that you want.

Nesting Components

Once you have assembled components into an application, often that application itself can be turned into a component. An earlier example showed an Animation component dropped onto a panel to display an animation.

Most of the code for the example was created through menu selections. The component was customized primarily by specifying the names of animation files to be displayed from the disk.

The mouseDrag event handler was overridden through the browser for detailed customization.

Once custom properties and behavior are defined, they can be preserved by making a new Bean out of the customized instance.

This is usually accomplished by serializing the customized Bean using Java's built-in support for object serialization. Most builder tools provide a way to turn compound components into custom Beans.

In this example you select the Animation Bean, and issue a menu command to turn it into a Bean component.



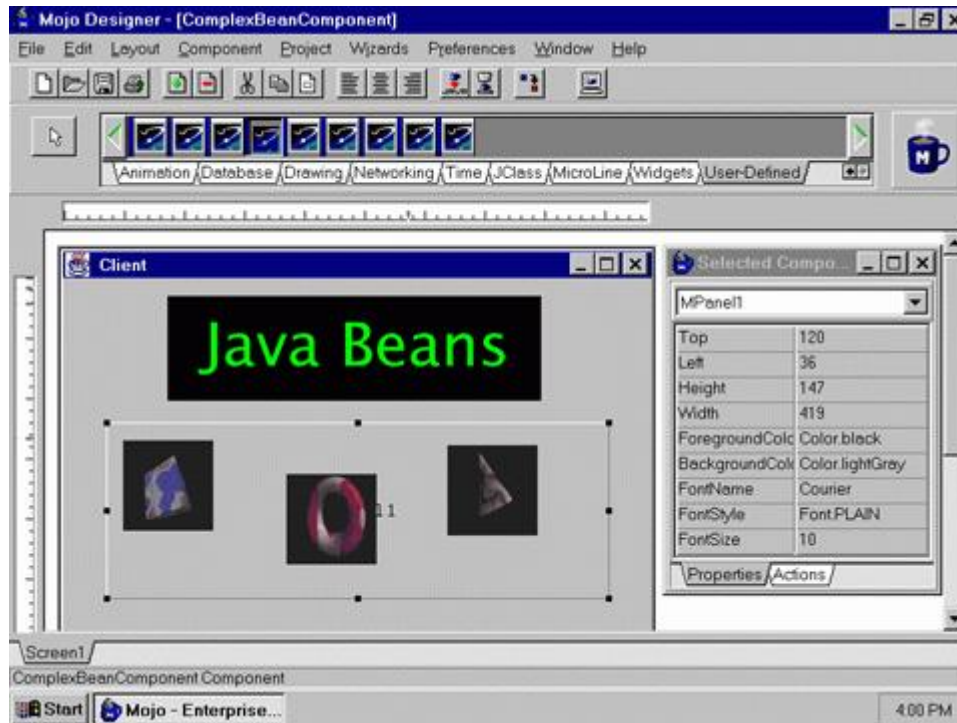
The builder generates code to turn the assembly (property and behavior definitions) into a Bean. The bean is added to the component palette of the builder tool.

Alternately the bean can be packaged in a JAR archive file for sale, or distribution to clients.

You could take the new custom Bean and use it to compose a compound assembly, creating yet another custom Bean.

For example, you could group three instance of a MyAnimationBean on a panel and save the whole assembly as a single new Bean.

The compound MyThreeBeans custom component is added to your builder's component palette as a single Bean.



5. WRITE IN DETAIL ABOUT NETWORK PROGRAMMING IN JAVA.

The term *network programming* refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.

The java.net package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing you to write programs that focus on solving the problem at hand.

The java.net package provides support for the two common network protocols:

TCP: TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.

UDP: UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

SOCKET PROGRAMMING:

Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

When the connection is made, the server creates a socket object on its end of the communication. The client and server can now communicate by writing to and reading from the socket.

The `java.net.Socket` class represents a socket, and the `java.net.ServerSocket` class provides a mechanism for the server program to listen for clients and establish connections with them.

The following steps occur when establishing a TCP connection between two computers using sockets:

The server instantiates a `ServerSocket` object, denoting which port number communication is to occur on.

The server invokes the `accept()` method of the `ServerSocket` class. This method waits until a client connects to the server on the given port.

After the server is waiting, a client instantiates a `Socket` object, specifying the server name and port number to connect to.

The constructor of the `Socket` class attempts to connect the client to the specified server and port number. If communication is established, the client now has a `Socket` object capable of communicating with the server.

On the server side, the `accept()` method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using I/O streams. Each socket has both an `OutputStream` and an `InputStream`. The client's `OutputStream` is connected to the server's `InputStream`, and the client's `InputStream` is connected to the server's `OutputStream`.

TCP is a two way communication protocol, so data can be sent across both streams at the same time. There are following usefull classes providing complete set of methods to implement sockets.

SERVERSOCKET CLASS METHODS:

The `java.net.ServerSocket` class is used by server applications to obtain a port and listen for client requests

The `ServerSocket` class has four constructors:

1. **`ServerSocket(int port)`** - Attempts to create a server socket bound to the specified port. An exception occurs if the port is already bound by another application
2. **`ServerSocket(int port, int backlog)`** - Similar to the previous constructor, the `backlog` parameter specifies how many incoming clients to store in a wait queue.
3. **`ServerSocket(int port, int backlog, InetAddress address)`** - Similar to the previous constructor, the `InetAddress` parameter specifies the local IP address to bind to. The `InetAddress` is used for servers that may have multiple IP addresses, allowing the server to specify which of its IP addresses to accept client requests on

4. **ServerSocket()**- Creates an unbound server socket. When using this constructor, use the bind() method when you are ready to bind the server socket

If the ServerSocket constructor does not throw an exception, it means that your application has successfully bound to the specified port and is ready for client requests.

Here are some of the common methods of the ServerSocket class:

1. **getLocalPort()**- Returns the port that the server socket is listening on. This method is useful if you passed in 0 as the port number in a constructor and let the server find a port for you.
2. **accept()**- Waits for an incoming client. This method blocks until either a client connects to the server on the specified port or the socket times out, assuming that the time-out value has been set using the setSoTimeout() method. Otherwise, this method blocks indefinitely
3. **setSoTimeout(int timeout)**- Sets the time-out value for how long the server socket waits for a client during the accept().
4. **bind(SocketAddress host, int backlog)**- Binds the socket to the specified server and port in the SocketAddress object. Use this method if you instantiated the ServerSocket using the no-argument constructor.

When the ServerSocket invokes accept(), the method does not return until a client connects. After a client does connect, the ServerSocket creates a new Socket on an unspecified port and returns a reference to this new Socket. A TCP connection now exists between the client and server, and communication can begin.

SOCKET CLASS METHODS:

The **java.net.Socket** class represents the socket that both the client and server use to communicate with each other. The client obtains a Socket object by instantiating one, whereas the server obtains a Socket object from the return value of the accept() method.

The Socket class has five constructors that a client uses to connect to a server:

1. **Socket(String host, int port)**- This method attempts to connect to the specified server at the specified port. If this constructor does not throw an exception, the connection is successful and the client is connected to the server.
2. **Socket(InetAddress host, int port)** - This method is identical to the previous constructor, except that the host is denoted by an InetAddress object.
3. **Socket(String host, int port, InetAddress localAddress, int localPort)**- Connects to the specified host and port, creating a socket on the local host at the specified address and port.
4. **Socket(String host, int port, InetAddress localAddress, int localPort)** - Connects to the specified host and port, creating a socket on the local host at the specified address and port.
5. **Socket()**- Creates an unconnected socket. Use the connect() method to connect this socket to a server.

When the Socket constructor returns, it does not simply instantiate a Socket object but it actually attempts to connect to the specified server and port.

Some methods of interest in the Socket class are listed here. Notice that both the client and server have a Socket object, so these methods can be invoked by both the client and server.

1. **connect(SocketAddress host, int timeout)** - This method connects the socket to the specified host. This method is needed only when you instantiated the Socket using the no-argument constructor.
2. **getInetAddress()**- This method returns the address of the other computer that this socket is connected to.
3. **getPort()**- Returns the port the socket is bound to on the remote machine.
4. **getLocalPort()**- Returns the port the socket is bound to on the local machine.
5. **getRemoteSocketAddress()**-Returns the address of the remote socket.
6. **getInputStream()** - Returns the input stream of the socket. The input stream is connected to the output stream of the remote socket.
7. **getOutputStream()** - Returns the output stream of the socket. The output stream is connected to the input stream of the remote socket
8. **close()** - Closes the socket, which makes this Socket object no longer capable of connecting again to any server

INETADDRESS CLASS METHODS:

This class represents an Internet Protocol (IP) address. Here are following useful methods which you would need while doing socket programming:

1. **getByAddress(byte[] addr)**- Returns an InetAddress object given the raw IP address
2. **getByAddress(String host, byte[] addr)**- Create an InetAddress based on the provided host name and IP address.
3. **getByName(String host)**- Determines the IP address of a host, given the host's name.
4. **getHostAddress()** - Returns the IP address string in textual presentation.
5. **getHostName()** - Gets the host name for this IP address.
6. **getLocalHost()**- Returns the local host.
7. **toString()**- Converts this IP address to a String.

CREATING TCP SERVERS

To create a TCP server, do the following:

1. Create a ServerSocket attached to a port number.

```
ServerSocket server = new ServerSocket(port);
```

2. Wait for connections from clients requesting connections to that port.

```
// Block on accept()
Socket channel = server.accept();
```

You'll get a Socket object as a result of the connection.

3. Get input and output streams associated with the socket.

```
out = new PrintWriter(channel.getOutputStream());
reader = new InputStreamReader(channel.getInputStream());
in = new BufferedReader (reader);
```

Now you can read and write to the socket, thus, communicating with the client.

```
String data = in.readLine();
out.println("Hi!");
```

When a server invokes the accept() method of the ServerSocket instance, the main server thread blocks until a client connects to the server; it is then prevented from accepting further client connections until the server has processed the client's request

Creating TCP Clients

To create a TCP client, do the following:

1. Create a Socket object attached to a remote host, port.

```
Socket client = new Socket(host, port);
When the constructor returns, you have a connection.
```

2. Get input and output streams associated with the socket.

```
out = new PrintWriter(client.getOutputStream());
reader = new InputStreamReader(client.getInputStream());
in = new BufferedReader (reader);
```

Now you can read and write to the socket, thus, communicating with the server.

```
out.println("Hello!!!");
String data = in.readLine();
```

EXAMPLE PROGRAM:

CHAT SERVER:

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

```

class TcpOneWayChatServer
{
    public static void main(String a[])throws IOException
    {
        ServerSocket ss = new ServerSocket(8000);
        Socket s=ss.accept();

        BufferedReader keyIn = new BufferedReader(new InputStreamReader(System.in));
        PrintStream socOut = new PrintStream(s.getOutputStream());

        while(true)
        {
            System.out.print("Message: ");
            String str = keyIn.readLine();
            if(str.equals("bye"))
            {
                break;
            }
            socOut.println(str);
        }
    }
}

```

CHAT CLIENT

```

import java.io.*;
import java.net.*;

class TcpOneWayChatClient
{
    public static void main(String args[])throws IOException
    {
        Socket c = new Socket("localhost", 8000);
        BufferedReader socIn=new BufferedReader(new InputStreamReader(c.getInputStream()));
        String str;
        while(true)
        {
            str = socIn.readLine();
            System.out.println("Message Received: " + str);
        }
    }
}

```

8. WRITE IN DETAIL ABOUT REMOTE METHOD INVOCATION

The Remote Method Invocation (RMI) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM.

The RMI provides remote communication between the applications using two objects stub and skeleton.

RMI uses stub and skeleton object for communication with the remote object. A **remote object** is an object whose method can be invoked from another JVM.

STUB AND SKELETON OBJECTS:

STUB

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

SKELETON

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.
4. In the Java 2 SDK, a stub protocol was introduced that eliminates the need for skeletons.



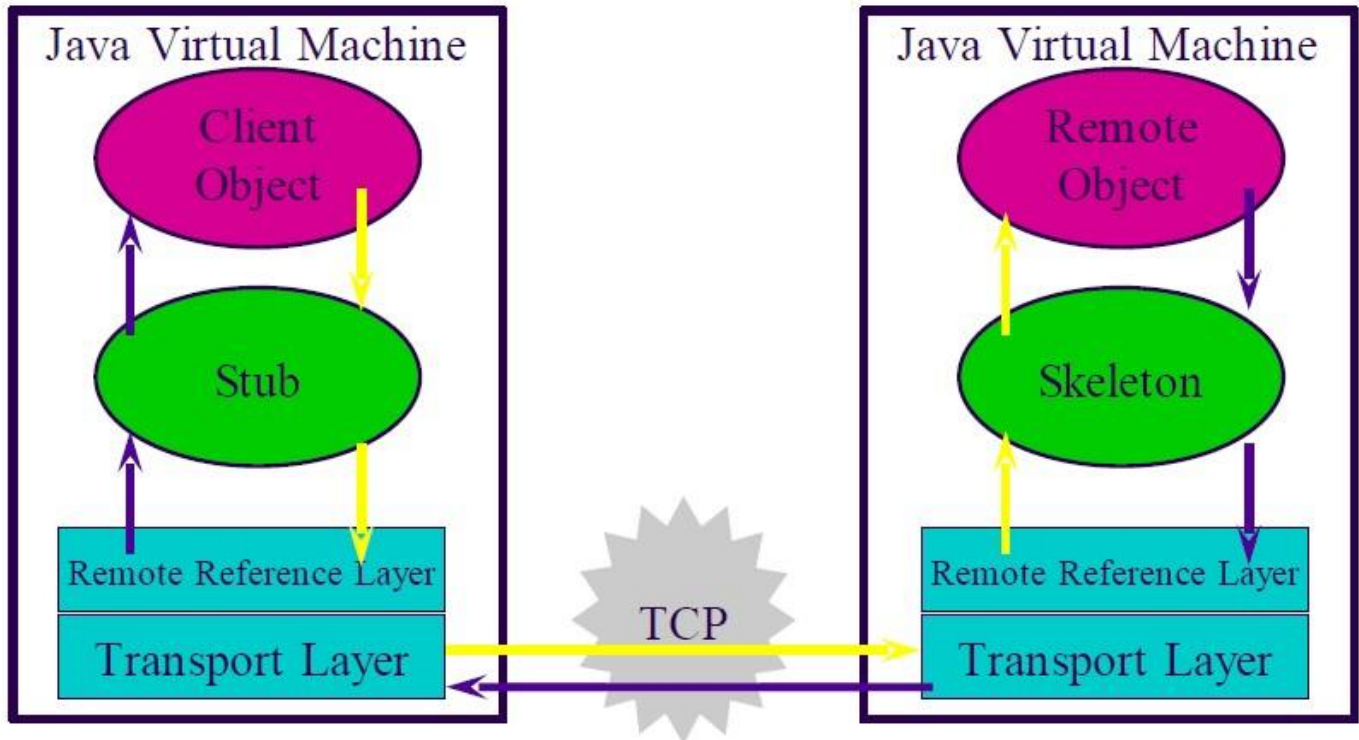
REQUIREMENTS FOR THE DISTRIBUTED APPLICATIONS

If any application performs these tasks, it can be distributed application.

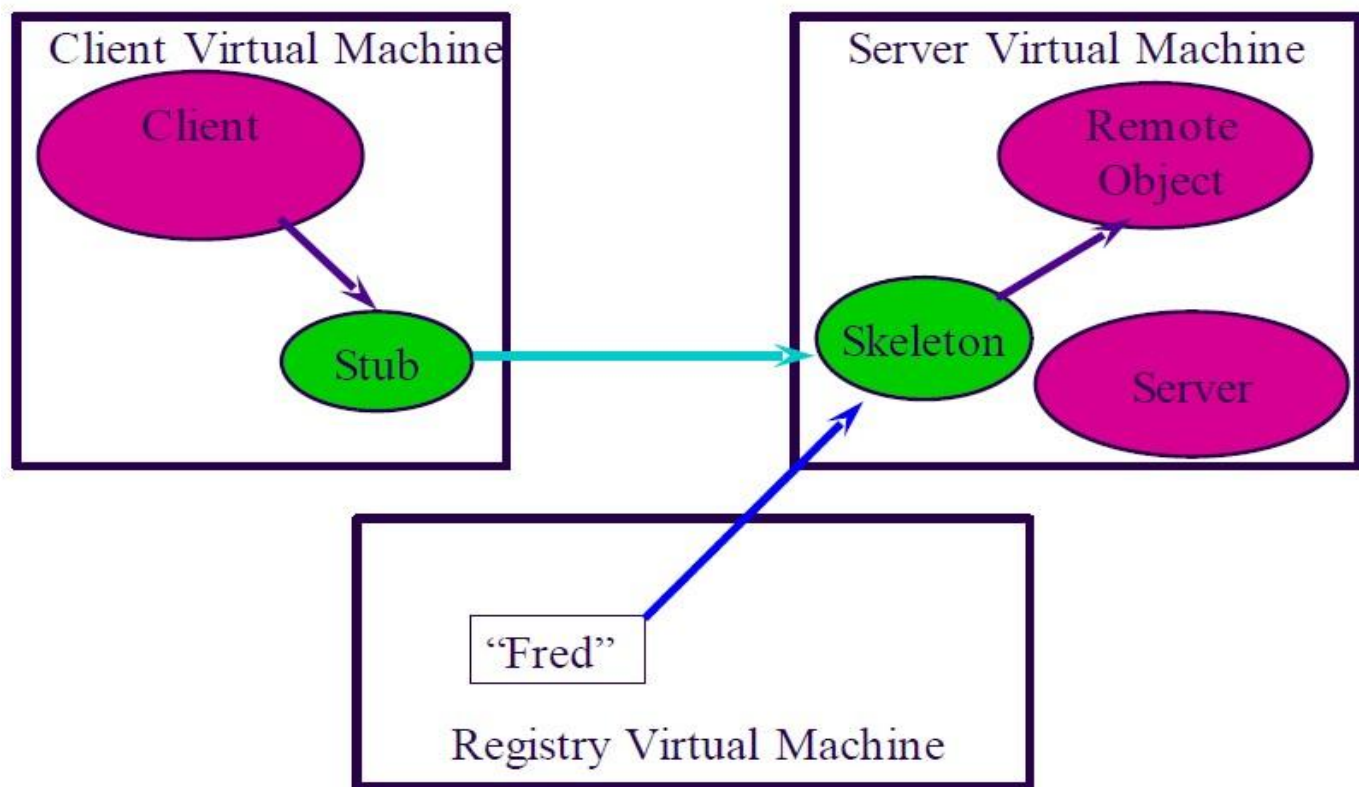
1. The application need to locate the remote method
2. It need to provide the communication with the remote objects, and
3. The application need to load the class definitions for the objects.

The RMI application have all these features, so it is called the distributed application.

RMI LAYERS

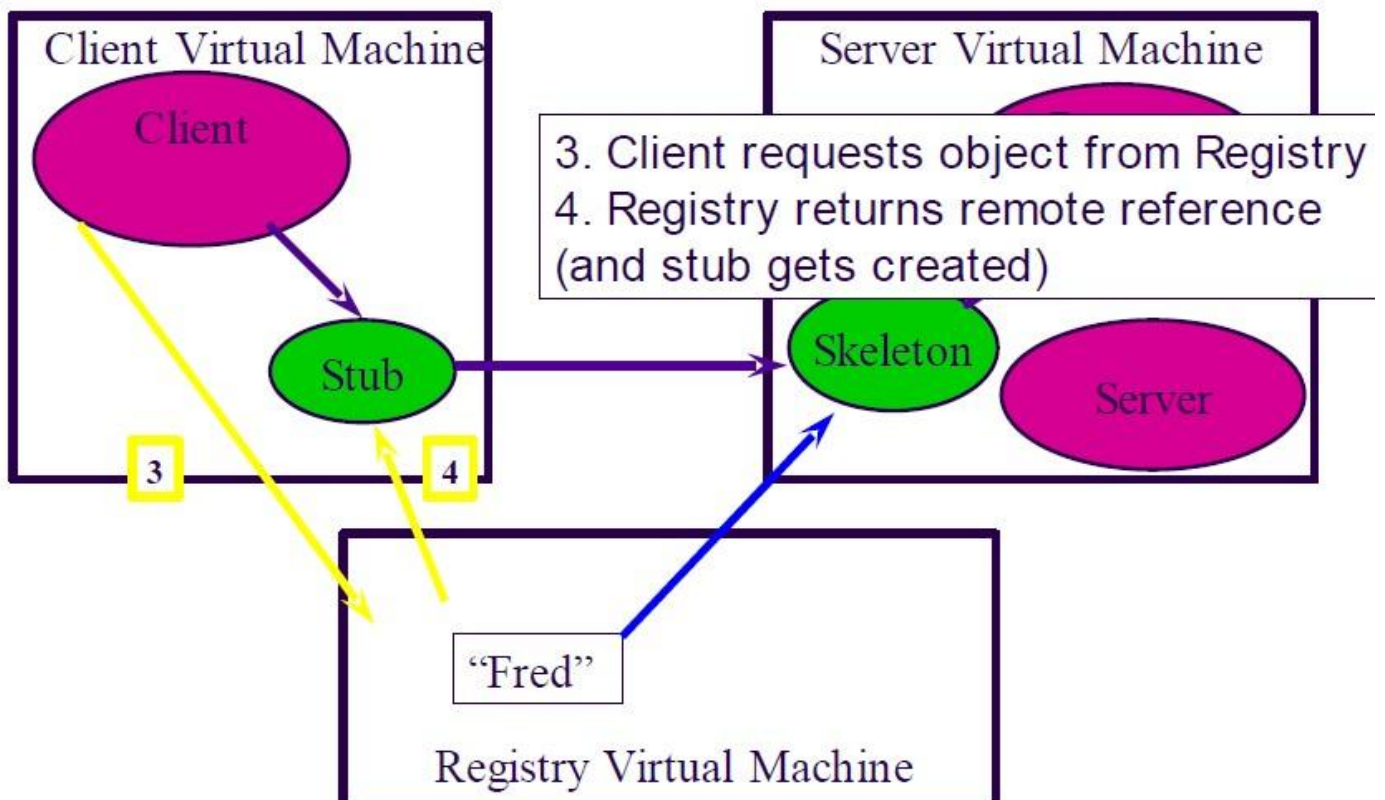
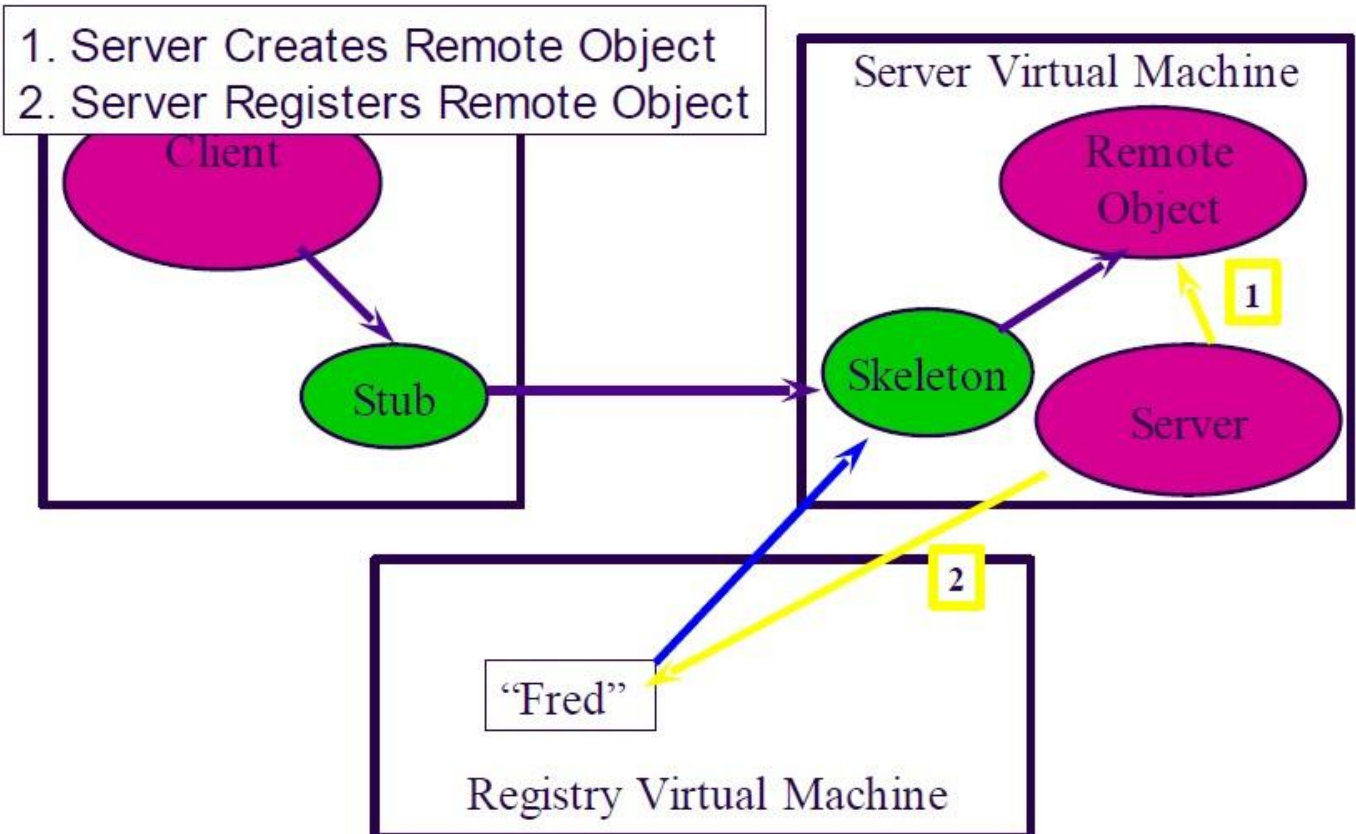


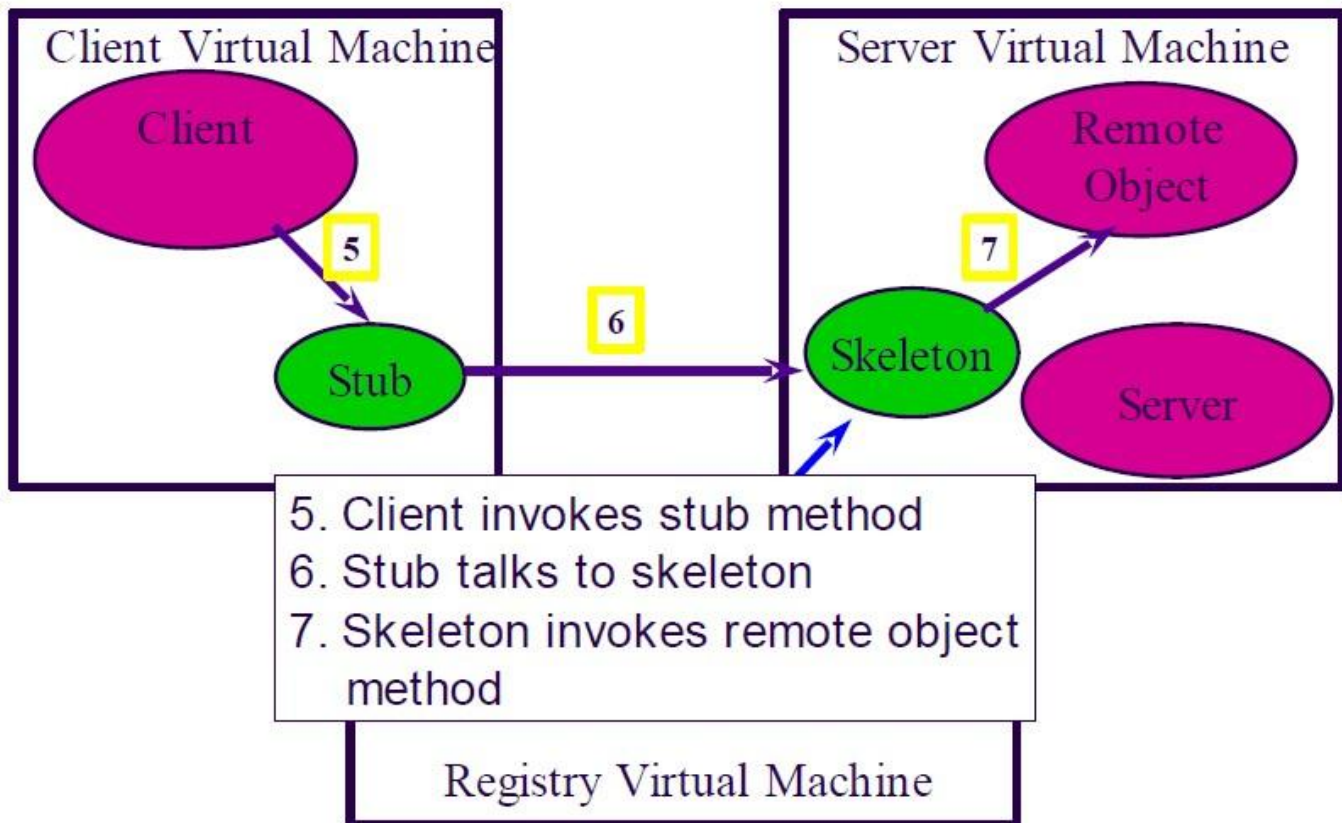
RMI SYSTEM ARCHITECTURE



RMI FLOW

1. Server Creates Remote Object
2. Server Registers Remote Object





STEPS TO WRITE THE RMI PROGRAM

There are 5 steps to follow

Server Side

1. Create the remote interface
2. Provide the implementation of the remote interface that extends `UnicastRemoteObject`
3. Create object for implementation class and register the object with the rmi registry

Client side

4. Create class in client side and get the remote reference by lookup the registry in the remote system
5. Using the remote reference, invoke the remote method

1. Create the remote interface

```
public interface FactInterface extends Remote
{
    public int fact(int n) throws RemoteException;
}
```

2. Provide the implementation of the remote interface that extends `UnicastRemoteObject`

```
public class FactImplementation extends UnicastRemoteObject implements FactInterface
{
    public FactImplementation() throws RemoteException
```

```
{
}
public int fact(int n) throws RemoteException
{
    int f = 1;
    for(int i = 1; i <= n; i++)
    {
        f = f * i;
    }
    return f;
}
}
```

3. Create object for implementation class and register the object with the rmi registry

```
FactImplementation fi = new FactImplementation();
Naming.rebind("server", fi);
System.out.println("Server ready");
```

4. Create class in client side and get the remote reference by lookup the registry in the remote system

```
FactInterface serv = (FactInterface) Naming.lookup("rmi://"127.0.0.1"/server");
```

5. Using the remote reference, invoke the remote method

```
int fact = serv.fact(5);
```