UNIT I : Object oriented thinking :- Need for OOP paradigm, A way of viewing world – Agents, responsibility, messages, methods, classes and instances, class hierarchies (Inheritance), method binding, overriding and exceptions, summary of OOP concepts.
Java Basics :- History of Java, Java buzzwords, data types, variables, constants, scope and life time of variables, operators, expressions, control statements, type conversion and casting, simple java programs, concepts of classes, objects, arrays, strings, constructors, methods, access control, this keyword, garbage collection, overloading methods and constructors, parameter passing, BufferedReader class, Scanner class, StringTokenizer class, inner class.

Need for OOP paradigm

• The object oriented paradigm is a methodology for producing reusable software components.

• The object-oriented paradigm is a programming methodology that promotes the efficient design and development of software systems using reusable components that can be quickly and safely assembled into larger systems.

• Object oriented programming has taken a completely different direction and will place an emphasis on object s and information. With object oriented programming, a problem will be broken down into a number of units .these are called objects .The foundation of OOP is the fact that it will place an emphasis on objects and classes. There are number of advantages to be found with using the OOP paradigm, and some of these are OOP paradigm.

• Object oriented programming is a concept that was created because of the need to overcome the problems that were found with using structured programming techniques. While structured programming uses an approach which is top down, OOP uses an approach which is bottom up.

• A **paradigm** is a way in which a computer language looks at the problem to be solved. We divide computer languages into four paradigms: *procedural*, *object-oriented*, *functional* and *declarative*

• A paradigm shift from a function-centric approach to an object-centric approach to software development.

• A program in a procedural paradigm is an active agent that uses passive objects that we refer to as data or data items.

• The basic unit of code is the **class** which is a template for creating run-time objects.

• Classes can be composed from other classes. For example, Clocks can be constructed as an aggregate of Counters.

• The object-oriented paradigm deals with **active objects** instead of passive objects. We encounter many active objects in our daily life: a vehicle, an automatic door, a dishwasher and so on. The actions to be performed on these objects are included in the object: the objects need only to receive the appropriate stimulus from outside to perform one of the actions.

• A file in an object-oriented paradigm can be packed with all the procedures called **methods** in the object-oriented paradigm—to be performed by the file: printing, copying, deleting and so on. The program in this paradigm just sends the corresponding request to the object.

• Java provides automatic garbage collection, relieving the programmer of the need to ensure that unreferenced memory is regularly deallocated.

Object Oriented Paradigm – Key Features

• Encapsulation
• Abstraction
• Inheritance
• Polymorphism

A Way of viewing World-

Agents

• A **Java agent** is a regular Java class which follows a set of **strict conventions**. The agent class must implement a ***public static premain method*** similar in principle to the main application entry point. *After the Java Virtual Machine (JVM) has initialized, each premain method will be called in the order the agents were specified, then the real application main method will be called.* There are agent development toolkits and agent programming languages.

• The **Agent Identity** class defines agent identity. An instance of this class uniquely identifies an agent. Agents use this information to identify the agents with whom they are interested in collaborating.

• The **Agent Host** class defines the agent host. An instance of this class keeps track of every agent executing in the system. It works with other hosts in order to transfer agents.

• The **Agent** class defines the agent. An instance of this class exists for each agent executing on a given agent host.

• ***OOP uses an approach of treating a real world agent as an object***.

• Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data.

• An object-oriented program can be characterized as data controlling access to code by switching the controlling entity to data.

Responsibility

• In object-oriented design, the chain-of-responsibility pattern is a design pattern consisting of a source of command objects and a series of processing objects.

• Each processing object contains logic that defines the types of command objects that it can handle; the rest are passed to the next processing object in the chain. A mechanism also exists for adding new processing objects to the end of this chain.

• Primary motivation is the need for a platform-independent (that is, architecture- neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls.

• Objects with clear responsibilities.

• Each class should have a clear responsibility.

• If you can't state the purpose of a class in a single, clear sentence, then perhaps your class structure needs some thought.

• In **object-oriented programming,** the **single responsibility principle** states that every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility.

Messages

• Message implements the Part interface. Message contains a set of attributes and a "content".

• Message objects are obtained either from a Folder or by constructing a new Message object of the appropriate subclass. Messages that have been received are normally retrieved from a folder named "INBOX".

• A Message object obtained from a folder is just a lightweight reference to the actual message. The Message is 'lazily' filled up (on demand) when each item is requested from the message.

• Note that certain folder implementations may return Message objects that are pre-filled with certain user-specified items. To send a message, an appropriate subclass of Message (e.g., Mime Message) is instantiated, the attributes and content are filled in, and the message is sent using the Transport. Send method.

• We all like to use programs that let us know what's going on. Programs that keep us informed often do so by displaying status and error messages.

• These messages need to be translated so they can be understood by end users around the world.

• The Section discusses translatable text messages. Usually, you're done after you move a message String into a Resource Bundle.

• If you've embedded variable data in a message, you'll have to take some extra steps to prepare it for translation.

Methods

• The only required elements of a method declaration are the method's **return type, name, a pair of parentheses, (), and a body between braces, {}.**

• Two of the components of a method declaration comprise the ***method signature*** - the method's name and the parameter types.

• More generally, method declarations have six components, in order:

• **Modifiers** - such as public, private, and others you will learn about later.

The **return type** - the data type of the value returned by the method, but void method does not return a value to calling method.

• The **method name** - the rules for field names apply to method names as well, but the convention is a little different.

• The **parameter list in parenthesis** - a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses(). **Ex:** int add(int a,int b) { } or int add() { }

• The method body, enclosed between braces - the method's code, including the declaration of local variables, goes here. { //body of amethod }

Naming a Method

Although a method name can be any **legal identifier**, code conventions restrict method names. By convention, method names should be a verb in lowercase or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc. In multiword names, the first letter of each of the second and following words should be capitalized. Here are some examples:

run
runFast
getBackground
getFinalData

Typically, a method has a unique name within its class. However, a method might have the same name as other methods due to ***method overloading***.

Overloading Methods

• The Java programming language supports *overloading* methods, and Java can distinguish between methods with different ***method signatures***. This means that methods within a class can have the same name if they have different parameter lists (there are some qualifications to this that will be discussed in the lesson titled "Interfaces and Inheritance").

• In the Java programming language, you can use the same name for all the drawing methods but pass a different argument list to each method. Thus, the data drawing class might declare four methods named draw, each of which has a different parameter list.

• Overloaded methods are differentiated by the number and the type of the arguments passed into the method.

• You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

• The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

• Overloaded methods should be used sparingly, as they can make code much less readable.

Classes

• In object-oriented terms, we say that your bicycle is an *instance* of the *class of objects* known as bicycles. A *class* is the blueprint from which individual objects are created.

• Java classes contain ***fields*** **and** ***methods***. A field is like a C++ data member, and a method is like a C++ member function. In Java, each class will be in its own .java file. Each field and method has an *access level*:

• **private:** accessible only in this class.

• **(package):** accessible only in this package.

• **protected:** accessible only in this package and in all subclasses of this class.

• **public:** accessible everywhere this class is available.

Each class has one of two possible access levels:

• **(package):** class objects can only be declared and manipulated by code in this package.
• **Public:** class objects can be declared and manipulated by code in any package.

• **Object:** Object-oriented programming involves *inheritance*. **In Java, all classes (built-in or user-defined) are (implicitly) subclasses of Object**. Using an array of Object in the List class allows any kind of Object (an instance of any class) to be stored in the list. However, primitive types (int, char, etc) cannot be stored in the list.

• A method should be made static when it does not access any of the non-static fields of the class, and does not call any non-static methods.

• Java class objects exhibit the properties and behaviors defined by its class. A class can contain fields and methods to describe the behavior of an object. Current states of a class_s corresponding object are stored in the object_s instance variables.

Creating a class:

A class is created in the following way

```
Class <class name>
 {
   Member variables;
   Methods;
}
```

• An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. This lesson explains how state and behavior are represented within an object, introduces the concept of data encapsulation, and explains the benefits of designing your software in this manner.

Class Variables – Static Fields

We use class variables also know as ***Static fields*** when we want to share characteristics across all objects within a class. When you declare a field to be static, only a single instance of the associated variable is created common to all the objects of that class. Hence when one object changes the value of a class variable, it affects all objects of the class. We can access a class variable by using the name of the class, and not necessarily using a reference to an individual object within the class. Static variables can be accessed even though no objects of that class exist. It is declared using ***static keyword***.

Class Methods – Static Methods

Class methods, similar to Class variables can be invoked without having an instance of the class. Class methods are often used to provide global functions for Java programs.

For example, methods in the java.lang.Math package are class methods. You cannot call nonstatic methods from inside a static method. Bundling code into individual software objects provides a number of benefits, including:

• **Modularity:** The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.

• **Information-hiding**: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.

• **Code re-use**: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.

• **Pluggability and debugging ease**: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace *it*, not the entire machine.

An instance or an object for a class is created in the following way

<class name> <object name>=new <constructor>();

Encapsulation:

• *Encapsulation* is the mechanism that ***binds together code and the data*** it manipulates, and keeps both safe from outside interference and misuse.

• One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper.

• Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.

• To relate this to the real world, consider the automatic transmission on an automobile.

• It encapsulates hundreds of bits of information about your engine, such as how much we are accelerating, the pitch of the surface we are on, and the position of the shift.

• The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects.

Polymorphism:

Polymorphism (from the Greek, meaning ―many forms‖) is a feature that allows one interface to be used for a general class of actions (One in many forms).

• The specific action is determined by the exact nature of the situation. Consider a stack (which is a last-in, first-out list). We might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs.

• In Java we can specify a general set of stack routines that all share the same names.

More generally, the concept of polymorphism is often expressed by the phrase - one interface, multiple methods. This means that it is possible to design a generic interface to a group of related activities.

• This helps reduce complexity by allowing the same interface to be used to specify a *general class of action.*

• Polymorphism allows us to create clean, sensible, readable, and resilient code.

Inheritance or class Hierarchies:

• Object-oriented programming allows classes to ***inherit*** commonly used state and behavior from other classes. Different kinds of objects often have a certain amount in common with each other.

• In the Java programming language, each class is allowed to have one direct *superclass*, and each *superclass* has the potential for an unlimited number of *subclasses*.

• Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear). Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio. In this example, Bicycle now becomes the *super class* of Mountain Bike, Road Bike, and Tandem Bike.

• The syntax for creating a subclass is simple. At the beginning of your class declaration, use the **extends keyword**, followed by the name of the class to inherit from:

```
class <sub class> extends <super class>
{
// new fields and methods defining a sub class would go here
}
```

The different types of inheritance are:

1. Single level Inheritance.
2. Multilevel Inheritance.
3. Hierarchical inheritance.
4. Multiple inheritance.
5. Hybrid inheritance.

Multiple, hybrid inheritance is not used in the way as other inheritances but it needs a special concept called **interfaces**.

Method Binding:

• Binding denotes association of a name with a class.

• Static binding is a binding in which the class association is made during compile time. This is also called as early binding.

• Dynamic binding is a binding in which the class association is not made until the object is created at execution time. It is also called as late binding.

Abstraction:

Abstraction in Java or Object oriented programming is a way to ***segregate/hiding*** implementation from interface and one of the five fundamentals along with Encapsulation, Inheritance, Polymorphism, Class and Object.

• An essential component of object oriented programming is Abstraction.

• Humans manage complexity through abstraction.

• For example people do not think a car as a set of tens and thousands of individual parts. They think of it as a well defined object with its own unique behavior.

• This abstraction allows people to use a car ignoring all details of how the engine, transmission and braking systems work.

• In computer programs the data from a traditional process oriented program can be transformed by abstraction into its component objects.

• A sequence of process steps can become a collection of messages between these objects. Thus each object describes its own behavior.

Overriding:

• In a class hierarchy when a sub class has the same name and type signature as a method in the super class, then the method in the subclass is said to override the method in the super class.

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

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

Exceptions:

• An exception is an ***abnormal condition*** that arises in a code sequence at run time. In other words an exception is a run time error.

• A java exception is an object that describes an exceptional condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.

Summary of OOP concepts

OOP) is a programming paradigm that represents concepts as **"objects"** that have data **fields** (attributes that describe the object) and associated procedures known as **methods**.

• Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

• Object-oriented programming is an approach to designing modular, reusable software systems.

• The goals of object-oriented programming are:

- ✓ Increased understanding
- ✓ Ease of maintenance
- ✓ Ease of evolution.
- ✓ Object orientation eases maintenance by the use of encapsulation and information hiding.

Object-Oriented Programming – Summary of Key Terms

Definitions of some of the key concepts in Object Oriented Programming (OOP)

Term	Definition
Abstract Data Type	A user-defined data type, including both attributes (its state) and methods (its behavior).
Aggregation	***Objects that are made up of other objects are known as aggregations***. The relationship is generally of one of **two types**: • **Composition** – the object is composed of other objects. This form of aggregation is a form of code reuse. *E.g. A Car is composed of Wheels, a Chassis and an Engine* • **Collection** – the object contains other objects. *E.g. a List contains several Items; A Set several Members*.
Attribute	A *characteristic of an object*. Collectively the attributes of an object describe its state. *E.g. a Car may have attributes of Speed, Direction, Registration Number and Driver.*
Class	The definition of objects of the same abstract data type. In Java ***class*** is the keyword used to define new types.
Encapsulation	The combining together of attributes (data) and methods (behavior/processes) into a single abstract data type with a public interface and a private implementation. This allows the implementation to be altered without affecting the interface.
Inheritance	Acquiring the properties from base class to derived class. The first class is often referred to the **base or parent class**. The child is often referred to as a **derived or sub-class**. Inheritance is one form of object-oriented code reuse. ***E.g.*** *Both Motorbikes and Cars are kinds of Motor Vehicles and therefore share some common attributes and behaviour but may add their own that are unique to that particular type.*
Interface	The behaviour that a class exposes to the outside world; its public face. Also called its contract_. In Java **interface is also a keyword** similar to class. However a Java interface contains no implementation: simply describes the behavior of an Object.
Member Variable	A *characteristic of an object* or See attribute
Method	The implementation of some behaviour of an object.
Message	The invoking of a method of an object. In OOPs objects send each other messages to achieve the desired behaviour.
Object	**An instance of a class**. Objects have state, identity and behaviour.
Overloading	Allowing the same method name to be used for more than one implementation. The different versions of the method vary according to their parameter lists. If this can be determined at compile time then static binding is used, otherwise dynamic binding is used to select the correct method as runtime.

Polymorphism	Generally, the ability of different classes of object to respond to the same message in different, class-specific ways. Polymorphic methods are used which have one name but different implementations for different classes. ***E.g.*** *Both the Plane and Car types might be able to respond to a turnLeft message. While the behaviour is the same,* ***the means of achieving it are specific to each type.***
Primitive Type	The basic types which are provided with a given object oriented programming language. *E.g. int, float, double, char, Boolean*
Static(Early) Binding	The identification at compile time of which version of a polymorphic method is being called. In order to do this the compiler must identify the class of an object.
Dynamic (Late) Binding	The identification at run time. When the class of an object cannot be identified at compile time, so dynamic binding must be used.

Advantage of OOPs over Procedure-oriented programming language

- OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.
- OOP provides data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere.
- OOP provides ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

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

Java Basics:

History of JAVA

Java is a high level programming Language. It was introduced by **–SUN Microsystems** in June 1995. It was developed by a team under ***James Gosling***. Its original name was **–OAK** and later renamed to **Java**. Java has become the standard for **Internet applications**.

Since the Internet consists of different types of computers and operating systems. A common language needed to enable computers. **To run programs that run on multiple plot forms**.

Java is Object-Oriented language built on C and C++. It derives its **syntax from C** and its **Object-Oriented features are influenced by C++**.

Java can be used to create **two types of programs**

- **Applications**
- **Applets**.

An **application** is a prg.that runs on the user's computers under the operating system.

An **Applet** is a small window based prg.that runs on HTML page using a java enabled web browser like internet Explorer, Netscape Navigator or an Applet Viewer.

The Java Programming Environment

Applets are java programs. That run as part of a web page and they depend on a web browser in order to run. Applications are programs that are stored on the user's system and they do not need a web browser in order to run.

The Java programming environment includes a number of development tools to develop applet and application. The development tools are part of the system known as **"Java Development Kit"** or **" JDK"**. The JDK include the following.

1. Packages that contain classes.
2. Compiler
3. Debugger

JDK (java development tool kit):

It is a software package from the sun micro systems where a new package JFC(java foundation classes) was available as a separate package

JDK provides tools in the bin directory of JDK and they are as follows:

Javac: Javac is the java compiler that translates the source code to byte codes. That is, it converts the source file. Namely the **.java file to .class file.**

Java: The java interpreter which runs applets and Applications by reading and interpreting the byte code files. That is, it executes the .class file.

Javadoc: Javadoc is the utility used to produce documentation for the classes from the java files.

JDB: JDB is a debugging tool.

The way these tools are applied to build and run application programs is as follows:

The source code file is created using a text editor and saved with a **.java (with Extension).** The source code is compiled using the **java compiler javac. This translates source code to byte codes.** The compiled code is executed using the java interpreter **java.**

JVM (JAVA VIRTUAL MACHINE)

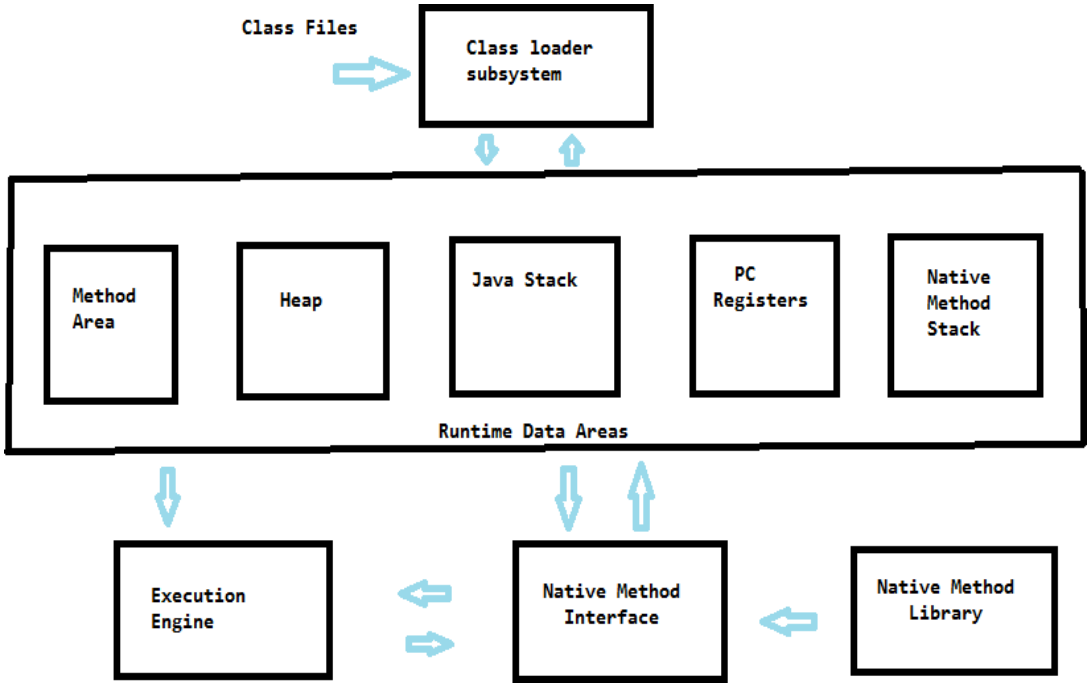
Java is both **compiled and an interpreted lang**. First the **java compiler** translates source code into the byte code instructions. In the next stage the **java interpreter** converts the byte code instructions to machine code. This machine within the computer‘ is known as the **“Java Virtual Machine” or JVM.**

The JVM can be thought as a mini operating system that forms a layer of abstraction where the underlying hardware. The **portability** feature of the **.class** file helps in the execution of the prg on any computer with the java virtual machine. This helps in implementing the **“write once and run anywhere” feature of java.**

JAVA RUNTIME ENVIRONMENT

JRE consists of the java virtual machine. The java plot form core classes and supporting files. It is the run time part of the java Development Kit. No compiler, no debugger, no tools.

Loding the .class files: Performed by the _ class loader‘ .
Verifying bytecode: Performed by the _ bytecode verifier‘ .
Executing the code : Performed by the runtime interpreter.



JIT (JUST – IN – TIME)

Just – In – Time (JIT) compiler is a program that runs java bytecode into instructions that can be sent directly to the processor. (Machine code).

Steps to execute a java prg:
? Open any editor such as notepad
? Type the source code
? Save it with **.java** extension (**File Name and class name must be same**)
? Compile it
? Run it

Steps to compile a java prg:
? Open Dos prompt
? Set the path **Ex: path=c:\jdk1.2.2\bin and press enter key**
? Move to your working directory/folder
? Compile the prg Ex: **javac filename.java**
? Run the prg Ex: **java filename**

IN THE JAVA PROGRAM:

public: It indicates that main() can be called outside the class.

static: It is an access specifier, which indicates that main() can be called directly without creating an object to the class.

void: It indicates that the method main() doesn‘ t return a value.

main(): It is a method which is an entry point into the java prg. When you run a java prg.main() is called first.

String args [ ]: String is a class, which belongs to java.lang package. It can be used as a string data type in java.

args[]: It is an array of string type. It is used to store command line args.

System: It is a class, which belongs to java.lang package.

out: It is an output stream object, which is a member of System class.

println(): It is a method supported by the output stream object ―out‖. It is used to display any kind of output on the screen. It gives a new line after printing the output.

print(): It is similar to println(). But doesn‘ t give a new line after printing the output.

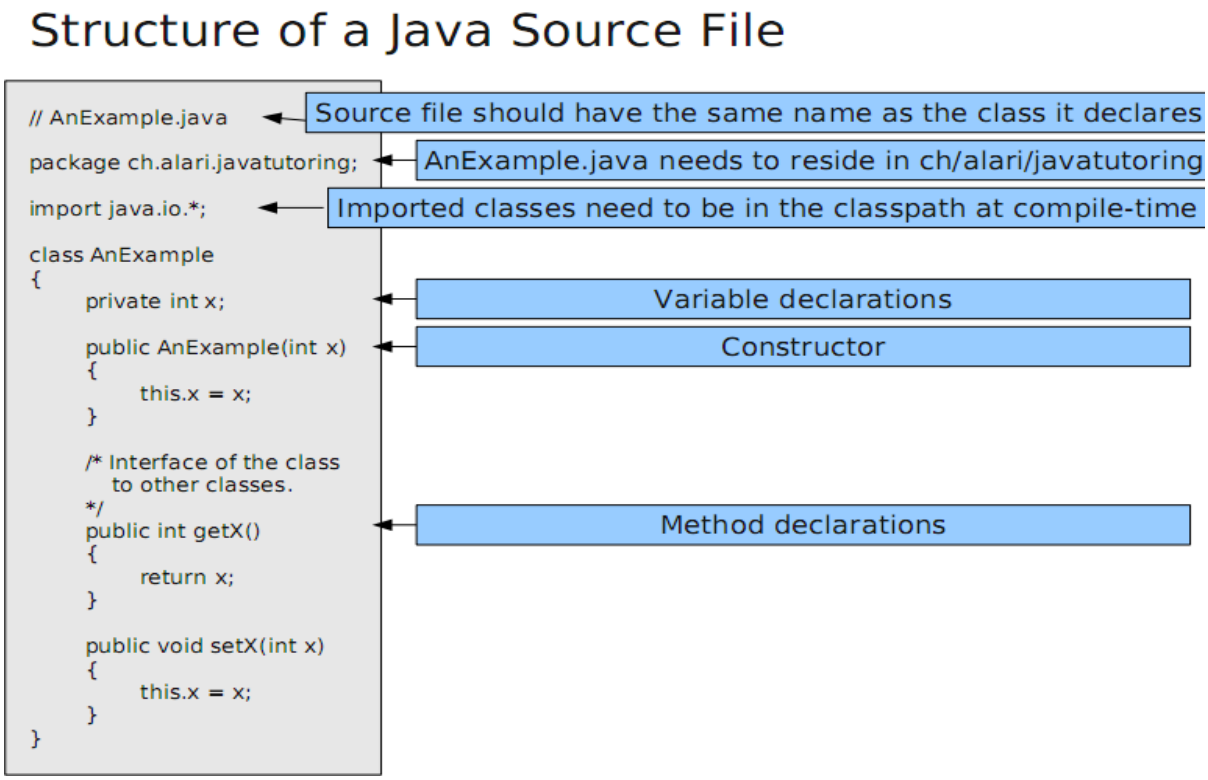
Java‟s Byte code:

The key that allows java to solve the both security and portability problems is that the output of a java compiler is not executable code rather it is byte code. Byte code is highly optimized set of instructions designed to be executed by java runtime systems, which is called JVM.

Byte codes are instructions that are generated for a virtual machine. A virtual machine is a program that processes these generalized instructions to ***machine specific code***.

Differences of JAVA from C++

- No typedefs, defines or preprocessors
- No header files
- No structures and unions
- No enums (enum class is there)
- No functions – only methods in classes
- No multiple inhehitance trough class (achieve using interface)
- No operator overloading (except _+‘ for string concatenation)
- No automatic type conversions (except for primitive types)
- No pointers



Java Buzzwords or Features of Java:

No discussion of the genesis of Java is complete without a look at the Java buzzwords. Although the fundamental forces that necessitated the invention of Java are portability and security, other factors also played an important role in mol ding the final form of the language. The key considerations were summed up by the Java team in the following list of buzzwords:

1. Simple: Java follows the syntax of C and Object Oriented principles of C++. It eliminates the complexities of C and C++. Therefore, Java has been made simple.

2. Object-Oriented: It is an Object-Oriented programming language. The object model in Java is simple. Although influenced by its procedures. It was designed to be source code compatible with any other language.

3. Plot form Independent: Plot form is the combination of operating system and microprocessor. Java programming works in all plot forms. It is achieved by JVM (Java Virtual Machine). The philosophy of java is ―Write Once, Run anywhere‖ (WORA).

4. Robust: Java is strictly a typed language. It has both a compiler and an interpreter. Compiler checks code at run time and garbage collection is taking care of by java automatically. Thus it is a robust language.

5. Secure: java developers have taken all care to make it a secure programming language.
For Ex. Java Applets are restricted from Disk I/O of local machine.

6. Distributed: Java is designed for the distributed environment of the Internet. It handles TCP/IP protocols. Java‘s remote method invocation (RMI) make distributed programming Possible.

7. Multithreaded: Java was designed to meet the real-world requirement. To achieve this, java supports multithreaded programming. It is the ability to run any things simultaneously.

8. Dynamic: That is run time. This makes it possible to dynamically link code in a safe and secure manner.

9. Architecture-Neutral

A central issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine.

Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java Virtual Machine (JVM) in an attempt to alter this situation. Their goal was ―write once; run anywhere, anytime, forever.

10. Interpreted and High Performance

Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java byte code. This code can be interpreted on any system that provides a Java Virtual Machine. It would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler.

Comments: There are 3 types of comments defined by java. Those are Single line comment, multilane comment and the third type is called documentation comment. This type of comment is used to produce an HTML file that documents your prg.

```
// this is single line comment.
/* this multiple
line comment*/
/** this documentation comment */
```

Key words:

Keywords are the words. Those have specifics meaning in the compiler. Those are called keywords. There are **49 reserved keywords** currently defined in the java language. These keywords cannot be used as names for a variable, class or method. Those are,

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

Java Is a Strongly Typed Language

It is important to state at the outset that Java is a strongly typed language. Indeed, part of Java‘s safety and robustness comes from this fact. Let‘s see what this means.

First, every variable has a type, every expression has a type, & every type is strictly defined. **Second,** all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility. There are no automatic coercions or conversions of conflicting types as in some languages.

The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

For example, in C/C++ you can assign a floating-point value to an integer as shown below
int n=12.345;
where in the above case the integer variable holds only the round part of the assigned fraction value as **n=12.**

In Java, you cannot. Also, **in C there is not necessarily** strong type-checking between a parameter and an argument. **In Java, there is**.

DATA TYPES:

The data, which gives to the computer in different types, are called **Data Types** *or* Storage representation of a variable is called **Data Type.** Java defines **8 types of data**: byte, short, int, long, float, double, char and Boolean.

These can be put in Four types:

Integer: this group includes **byte, short, int & long**, which are whole valued signed numbers.

Floating-point numbers: float & double, which represents numbers with fractional precision.

Character: This represents symbols in a character set, **like letters and numbers**.

Boolean: This is a special type for representing **true / false** values.

Name	*width(Bytes)*	*Range*
long	8	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
int	4	-2147483648 to +2147483647
short	2	-32,768 to +32,767
byte	1	-128 to 127
double	8	$-3.4*e^{308}$ to $3.4*e^{308}$
float	4	$-1.7*e^{38}$ to $1.7*e^{38}$
char	2	0 to 65,536
boolean	bit	0 or 1.

Examples of double and char

```
class FindSqrt
{
  public static void main(String args[])
  {
    double d1=25,d2=34;
    System.out.println("Sqrt of d1
                is:"+Math.sqrt(d1));
    System.out.println("Sqrt of d2
                is:"+Math.sqrt(d2));
  }
}
```

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

This program displays the following **output:**
ch1 and ch2: X Y

Examples of boolean

```
class BoolTest
 {
public static void main(String args[])
  {
boolean b;
b = false;
System.out.println("b is " + b);
b = true;
System.out.println("b is " + b);
if(b){
 System.out.println("This is executed."); }
b = false;
if(b) {
System.out.println("This is not executed.");}
System.out.println("10 > 9 is " + (10 > 9));
}
}
```

Output:

```
b is false
b is true
This is executed.
10 > 9 is true
```

Variables:

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime. In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

type identifier [= value][, identifier [= value] ...];

Ex: int a, b, c; // declares three ints, a, b, and c.
int d = 3, e, f = 5; // declares three more ints
byte z = 22; // initializes z.
double pi = 3.14159; // declares an approximation of pi.
char x = 'x'; // the variable x has the value 'x'.

Constant identifiers:

Final keyword is used for constants. Constant identifiers consist of all capital letters. Internal words are separated by an underscore (_).
Example: final double TAX_RATE = .05

The Scope and Lifetime of Variables:

All of the variables used till now have been declared at the start of the main( ) method. However, Java allows variables to be declared within any block. A block is begun with an opening curly brace and ended by a closing curly brace. **A block defines a scope**.

Most other computer languages define two general categories of scopes: **global and local**.
The scope defined by a method begins with its opening curly brace.

Objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.

To understand the effect of nested scopes, consider the following program:

```
// demonstrate block scope.
class Scope
 {
   public static void main(String args[])
   {
     int x; // code within main
     x = 10;
     if(x == 10)
      {    // start new scope
       int y = 20; // known only to this block

// x and y both known here.
System.out.print("x and y:" +x+ " " + y);
x = y * 2;
```

```
 }
y = 100; // Error! y not known here
  // x is still known here.
 System.out.print("\tx is " + x);
```

```
    }
}
```

Output: x and y: 10 20 x is 40

Types of variables: Java has 4 different kinds of variables

- local variables
- parameter variables
- class variables
- instance variables

Local variables:

A local variable will exists as long as the method in which they have been created is still running As soon as the method terminates, all the local variables inside the method are destroyed.

Example:

```
 class SomeClass
{
  public static void main(String args[])
  {
    double x;
    ……..
    int y;
}
     }
```

Local variables

Scope of local variables:

```
  class Scope2
  {
   public static void main(String args[ ]) {
   void show(){
     double r;
     r=3.14;
    }
   System.out.println(r );  //Causes error is not
    }                       accessible outside the block
  }
```

Non-overlapping (or disjoint) scopes:

It is possible to declare two variables with the same name in two different block scopes as shown below

```
class Scope {
  public static void main(String args[]) {
        - - - - - - -
    {      double r;
      {
          String r;
      }   }
  }
 }
```

Parameter variables:

A parameter variable is used to store information that is being passed from the location of the method call into the method that is called.

```
class ToolBox
{                                    a=1.0   b=2.0
  public static double min(double a,double b) //Parameter variables
  {    ……..    }
}
class MyProgram
{
```

```
 public static void main(String args[])    {
   double r;
   r=ToolBox.min(1.0,2.0);
    }   }
```

Life and scope of parameter variable:
The life time of a parameter variable is the entire body of the method
• A parameter variable behaves like a local variable that is defined at the Start of the method

Class variables:

Variables that are declared with the keyword static are called as class variables

Life of class variables: Class variables exists for the entire execution of the java program

Scope of class variables: The scope can be public

```
 class StaticData
{
  static int a=10,b=20; //Access these variables through class name, because of static.
}
class StaticDemo
{
     public static void main(String args[])
     {
           int sum=StaticData.a+StaticData.b;
           System.out.println("sum is"+sum);
      }     }
```

Instance variables:

Non static variables that are declared inside a class are called as instance variables.

```
class StaticData
{
  int a=10,b=20; //instance variables of StaticData class
}
```

Life and scope of Instance variables:
It is limited only the object specified upon the class

Literals

Integer Literals
Integers are probably the most commonly used type in the typical program. Any hole number value is an integer literal. Examples are 1, 2, 3, and 42.

Integer literals are classified into **three types** as
→ Decimal literal → Octal literal → Hexadecimal literal

Decimal literals
These are all decimal values, meaning they are describing a base 10 number. There are two other bases which can be used in integer literals

Example: int n=10;

Octal literals

Octal (base eight). Octal values are denoted **in Java by a leading zero**. Normal decimal numbers cannot have a leading zero.

Thus, the seemingly valid value 09 will produce an error from the compiler, since 9 is outside of **octal‟s 0 to 7 range**.

Example: int n=07;

Hexadecimal literals:

Hexadecimal (base 16), A more common base for numbers used by programmers is hexadecimal, which matches cleanly with modulo 8 word sizes, such as 8, 16, 32, and 64 bits. You signify a hexadecimal constant **with a leading zero-x**, (0x or 0X). The **range of a hexadecimal digit is 0 to 15**, so **A through F** (or a through f) are substituted for **10 through 15**.

Example: int n=0xfff;

Example Program:

```
class Literal
{
  public static void main(String args[])
  {
    int dec=10,oct=07,hex=0xff;
    System.out.print("decimal literal is:"+dec);
    System.out.println("octal literal is:"+oct);
    System.out.println("hexadecimal is:"+hex);
   }
}
```

Floating-Point Literals

Floating-point numbers represent decimal values with a fractional component.

Example: float f=12.346f (or) 12.346F;
double d=34.5678d (or) 34.5678D;

Boolean Literals

Boolean literals are simple. There are only two logical values that a boolean value can have, ***true and false***. The true literal in Java does not equal 1, nor does the false literal equal 0.

Character Literals

Characters in Java are indices into the **Unicode character set**. They are 16-bit values. A literal character is represented inside a pair of single quotes. All of the visible **ASCII** characters can be directly entered inside the quotes, such as ‟a‟, ‟z‟, and ‟@‟.

String Literals

String literals in Java are enclosing a sequence of characters between a pair of **double quotes**.

Ex: –Hello World‖ –two\nlines‖

–\‖This is in quotes\‖

Escape Sequence	Meaning
\n	new line
\t	horizontal tab
\v	vertical tab
\b	Backspace
\r	carriage return
\f	form feed
\a	Bell
\\	text literal
\' '	char literal
\" "	String literal
\ddd	Octal character(ddd)
\uxxxx	Hexa character (xxxx)

Java Naming conventions

Java **naming convention** is a rule to follow as you decide what to name your identifiers such as ***class, package, variable, constant, method*** etc. But, it is not forced to follow. So, it is known as *convention not rule*.

All the classes, interfaces, packages, methods and fields of java programming language are given according to java naming convention.

Advantage of naming conventions in java

By using standard Java naming conventions, you make your code easier to read for yourself and for other programmers. Readability of Java program is very important.

Name	Convention
class name	Should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc. **Ex: class MyClass**
interface name	Should start with uppercase letter and be an adjective **e.g. Runnable, Remote, ActionListener etc.**
method name	Should start with lowercase letter and be a verb **e.g. actionPerformed(), main(), print(), println() etc.**
variable name	Should start with lowercase letter **e.g. firstName, orderNumber etc.**
package name	Should be in lowercase letter **e.g. package java.lang;**
constants name	Should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc. **e.g. final double TAX_RATE = .05**

Camel Case in java naming conventions

Java follows **camelcase syntax** for naming the class, interface, method and variable. **If name is combined with two words, second word will start with uppercase letter always.**

e.g. *actionPerformed(), firstName, ActionEvent, ActionListener etc.*

OPERATORS:

Operator is a Symbol. Java provides a rich set of operators as

The Arithmetic Operators:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The arithmetic operators are: **+ - * / %**

The Relational Operators: It tells the relationship b/w two operands **e.g., a>b, a<b, a==b, a<=b , a>=b**

== (equal to),!= (not equal to),> (greater than),< (less than),>= (greater than or equal to),<= (less than or equal to)

The Assignment Operators (=): This operator is used to assign the value of a variable. **E.g. a=4**

Short hand assignment operator: += -= *= /= %= **E.g. a+=4 (a=a+4)**

The Bitwise Operators:

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; 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
```

Binary Left Shift Operator: << (left shift)
It is moved left by the number of bits specified by the right operand.
Example: A << 2 will give 240 which is 1111 0000

Binary Right Shift Operator: >> (right shift)
It is moved right by the number of bits specified by the right operand.
Example: A >> 2 will give 15 which is 1111

Shift right zero fill operator : >>> (zero fill right shift)
It is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.
Example: A >>>2 will give 15 which is 0000 1111

The Logical Operators:

Logical AND operator: && (logical and)
If **both** the operands are **non-zero**, then the condition becomes true.
Example (A && B) is false. **(0 && 1) or (1 && 0) or (0 && 0) is false & (1 && 1) is true**

Logical OR Operator: || (logical or)
If any of the two operands are non-zero, then the condition becomes true.
Example (A || B) is true. **(0 || 1) or (1 || 0) or (1 || 1) is true & (0 || 0) is false**

Logical NOT Operator: ! (logical not)
Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.
Example !(A && B) is true.

instanceof Operator: Type comparision 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)
public class Test
 {
    public static void main(String args[])
     {
       String name = "James";
       boolean result = name instanceof String;
       System.out.println( result );
      }
  }
```

Op: James

Unary operators: + - ++ - - ! sizeof

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
int x = (10>20)? 100 : 200

Operator Precedence:

Operator Precedence	
Operators	**Precedence**
Postfix	*expr++, expr--*
Unary	*++expr --expr +expr –expr* ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Expressions:

An expression is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language that evaluates to a single value.

```
int a = 0;
arr[0] = 100;
System.out.println("Element 1 at index 0: " + arr[0]);
int result = 1 + 2; // result is now 3
```

Statements

Statements are roughly equivalent to sentences in natural languages. A *statement* forms a complete unit of execution. The following types of expressions can be made into a statement by terminating the expression with a **semicolon (;)**. **Ex:** System.out.print(A+B);

Type Conversion and Casting:

We can assign ***a value of one type to a variable of another type.*** If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an int value to a long variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no conversion defined from double to byte.

But it is possible for conversion between **incompatible types**. To do so, you must use a **cast**, which performs an **explicit conversion between incompatible types**.

Java„s Automatic Conversions

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

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

When these two conditions are met, a **widening conversion** (Small data type to Big data type) takes place. **For example**, the int type is always large enough to hold all valid byte values, so *no explicit cast statement* is required.

For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, **the numeric types are not compatible with char or boolean**. Also, **char and boolean are not compatible with each other**.

Java also performs an **automatic type conversion** when storing a literal integer constant into variables of type byte, short, or long.

Casting Incompatible Types

The automatic type conversions are helpful, they will not fulfill all needs. **For example**, if we want to assign *an int value to a byte variable*. This conversion will **not be performed automatically**, because *a byte is smaller than an int*. This kind of conversion is sometimes called a **narrowing conversion**, since you are explicitly making the value narrower so that it will fit into the target type. To create a conversion between two incompatible types, you must use a ***cast***. *A cast is simply an explicit type conversion*.

It has this general form:

(target-type) value

Here, target-type specifies the desired type to convert the specified value to.

Example:

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

A different type of conversion will occur when a floating-point value is assigned to an integer type: **truncation**. As integers do not have fractional components so, when a floating-point value is assigned to an integer type, the fractional component is lost.

Example Program: Conversion.java

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

Output:

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

The Type Promotion Rules

Java defines several type promotion rules that apply to expressions. They are as follows:

First, all byte, short, and char values are promoted to int, as just described. Then, if one operand is a long, the whole expression is promoted to long. If one operand is a float, the entire expression is promoted to float. If any of the operands is double, the result is double.

Ex:

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

Let‟s look closely at the type promotions that occur in this line from the program:

double result = (f * b) + (i / c) - (d * s);

In the first subexpression, f * b, b is promoted to a float and the result of the subexpression is float. Next, in the subexpression i/c, c is promoted to int, and the result is of type int. Then, in d*s, the value of s is promoted to double, and the type of the sub expression is **double**.

Control Statements or Control Flow:

IF Statement

The IF statement is Java‘s conditional branch statement. It can be used to route program execution through two different paths.

The general form of the if statement:

if (condition) statement1;

Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a Boolean value. If the condition is true, then statement1 is executed.

IF –ELSE Statement

If the condition is true, then statement1 is executed. Otherwise statement2 is executed.

The general form of the if statement:

if (condition) statement1;
else statement2;

The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false.

```
void ifClause()
{
int a, b;
if(a < b)
a = 0;
else
b = 0;
}
```

Nested ifs

A nested if is an if statement that is the target of another if or else. Here is an example:

```
if(i == 10)
{
if(j < 20)
 a = b;
if(k > 100)
 c = d; // this if is
else
  a = c;        // associated with this else
}
else   a = d;          // this else refers to if(i == 10)
```

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested ifs is the if-else-if ladder. It looks like this:

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

Example: // Demonstrate if-else-if statements.

```
class IfElse
{
public static void main(String args[])
 {
     int month = 4; // April
     String season;
     if(month == 12 || month == 1 || month == 2)
     season = "Winter";
     else if(month == 3 || month == 4 || month == 5)
     season = "Spring";
     else if(month == 6 || month == 7 || month == 8)
     season = "Summer";
     else if(month == 9 || month == 10 || month == 11)
     season = "Autumn";
     else
     season = "Bogus Month";
     System.out.println("April is in the " + season + ".");
 }
}
```

Output:

April is in the Spring.

The switch Statement

Unlike if-then and if-then-else, the switch statement allows for any number of possible execution paths. A switch works with the byte, short, char, and int primitive data types. It also works with enumerated types.

Program: displays the name of the month, based on the value of month, using the switch statement.

```
class SwitchDemo
 {
   public static void main(String[] args) {
       int month = 8;
      switch (month) {
       case 1: System.out.println("January");
                break;
       case 2:
       System.out.println("February");
                break;
       case 3: System.out.println("March");
                break;
       case 4: System.out.println("April");
                break;
       case 5: System.out.println("May");
                break;
        case 6: System.out.println("June");
                 break;
        case 7: System.out.println("July");
                 break;
        case 8: System.out.println("August");
                 break;
        case 9: System.out.println("September");
                 break;
        case 10: System.out.println("October");
                  break;
        case 11: System.out.println("November");
                  break;
        case 12: System.out.println("December");
                 break;
        default: System.out.println("Invalid
       month.");
                 break;
           }
      }
    }
```

Output:"August"

Iteration Statements

Java‘s iteration statements are for, while, and do-while. These statements create what we commonly call loops. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met.

The while Statement

The while statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as:

```
  while (expression)
 {
    Statements // body of loop
}
```

Note: The while statement evaluates an expression, which must return a Boolean value.

Ex: class WhileDemo
```
{
    public static void main(String[] args)
   {
        int count = 1;
        while (count < 11)
        {
            System.out.println("Count is: " + count);
            count++;
        }
    }
}
```
Output: Count is:1 Count is:2 ..... Count is:10

do-while statement

The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its syntax can be expressed as:

```
do
{
    statement(s)
} while (expression);
```

The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once and while terminates with semicolon.

Ex Program:
```
class DoWhileDemo
 {
    public static void main(String[] args){
        int count = 1;
        do {
            System.out.println("Count is: " + count);
            count++;
        } while (count < 11);
    }
}
```
Output: Count is:1 Count is:2 ..... Count is:10

The for Statement

The for statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied.

The general form of the for statement can be expressed as follows:

```
for (initialization; termination; increment) {
   statement(s)
}
```

Ex: class ForTick {
```
    public static void main(String args[]) {
      for(int n=10; n>0; n--)
        System.out.println("tick " + n);
     }
     }
```

When using the for statement, we need to remember that

• The initialization expression initializes the loop; it's executed once, as the loop begins.
• When the termination expression evaluates to false, the loop terminates.
• The increment expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment or decrement a value.

// Without using the comma.

```
class Sample {
public static void main(String args[]) {
int a, b;
b = 4;
for(a=1; a<b; a++) {
System.out.println("a = " + a);
System.out.println("b = " + b);
b--;
}
} }
```

Output:

```
a = 1
b = 4
a = 2
b = 3
```

// Using the comma.

```
class Comma {
public static void main(String args[]) {
int a, b;
for(a=1, b=4; a<b; a++, b--) {
System.out.println("a = " + a);
System.out.println("b = " + b);
}
}
}
```

Output:

```
a = 1
b = 4
a = 2
b = 3
```

For-Each Version of the for Loop

A for-each style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. A for-each loop by using the keyword foreach,

Java adds the for-each capability by enhancing the for statement. The advantage of this approach is that no new keyword is required, and no pre existing code is broken. The for-each style of for is also referred to as the enhanced for loop.

The general form of the for-each version of the for is shown here:

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

//Using for loop

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int i=0; i < 10; i++) sum += nums[i];
```

//Using for-each loop

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x: nums) sum += x;
```

// Use a for-each style for on a 1D array.

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

Output:

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

// Use for-each style for on a 2D array

```
class ForEach3 {
public static void main(String args[]) {
int sum = 0;
int nums[ ][ ] = new int[3][5];
// give nums some values
for(int i = 0; i < 3; i++)
for(int j=0; j < 5; j++)
nums[i][j] = (i+1)*(j+1);
// use for-each to display & sum the values
for(int x[ ] : nums) {
for(int y : x) {
System.out.println("Value is: " + y);
sum += y;
} }
System.out.println("Summation: " + sum);
} }
```

Output:

```
Value is: 1    Value is: 2    Value is: 3
Value is: 4    Value is: 5

Value is: 2    Value is: 4    Value is: 6
Value is: 8    Value is: 10

Value is: 3    Value is: 6    Value is: 9
Value is: 12   Value is: 15
Summation: 90
```

Nested Loops

One loop may be inside another.

Ex: Nested.java

```
class Nested
{
public static void main(String args[])
 {
    int i, j;
     for(i=1; i<10; i++)
     {
       for(j=i; j<10; j++)
         {
           System.out.print(".");
         }
     System.out.println();
      }
}
}
```

Output:

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

Unconditional Control Statements

Java supports the following types of unconditional control statements.

1. break:

it is used to break any loop control statement or any switch control statement.

// Using break to exit a loop.

```
class BreakLoop {
public static void main(String args[]) {
for(int i=0; i<100; i++) {
if(i == 10) break; // terminate loop if i is 10
System.out.print(i+" "); }
System.out.print(" Loop complete.");
} }
```

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

2. continue

it is used to skip the current iteration.

// Using continue to skip the current iteration

```
class BreakLoop {
public static void main(String args[]) {
for(int i=0; i<10; i++) {
if(i == 5) continue;
System.out.print(i+" ");
}
System.out.print(" Loop complete.");
} }
```

Output: 0 1 2 3 4 6 7 8 9 Loop complete.

3.goto:

The main drawback of goto statement in the c language is it decrease the readability of a program to avoid that drawback. The goto in java is changed as goto break.

The syntax for the goto statement in java is:

```
Label:
{
    Statements;
    Break Label;
}
```

Example:

// Using break as a civilized form of goto.

```
class Break {
public static void main(String args[]) {
boolean t = true;
first:
{
second:
{
third:
{
```

```
System.out.println("Before the break.");
if(t) break second; // break out of second block
System.out.println("This won't execute");
}
System.out.println("This won't execute");
}
System.out.println("This is after second block.");
}
} }
```

Output:

Before the break.
This is after second block.

4. return:

The return control statement is used to jump the control from the called function to the calling function.

Classes and Objects:

A ***class*** is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical. A class in Java can contain:

- ***fields***
- ***methods***
- ***constructors***
- ***blocks***
- ***nested class and interface***

Declaring Member Variables

There are several kinds of variables:

- *Member variables* in a class—these are called **fields**.
- *Variables in a method* or block of code—these are called **local variables**.
- *Variables in method declarations*—these are called **parameters**.

Syntax to declare a class:

```
class <class_name>{
   field;
   method;
}
```

A **class is declared** by use of the **class keyword**

```
class classname
 {
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(parameter-list)
{
// body of method
}
type methodname2(parameter-list)
 {
// body of method
}
// ...
type methodnameN(parameter-list)
{
// body of method
}
}
```

The data, or variables, defined within a class are called **instance variables.** The code is contained within methods. Collectively, the methods and variables defined within a class are called **members of the class**. Variables defined within a class are called **instance variables.**	**Example:**
	Class demo {
	int x=1;;
	int y=2;
	float z=3;
	void display()
	{
	System.out.println(−values are:‖+x+--y+--z);
	}
	}

OBJECTS:

An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tangible and intangible). The example of intangible object is banking system.

An object has three characteristics:

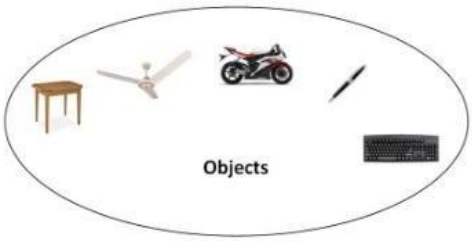
- **state:** represents data (value) of an object.
- **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally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

For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.

Object is an instance of a class. Class is a template or blueprint from which objects are created. So object is the instance(result) of a class.

Object Definitions:

- Object is a real world entity.
- Object is a run time entity.
- Object is an entity which has state and behavior.
- Object is an instance of a class.



New keyword in Java

The new keyword is used to allocate memory at run time. All objects get memory in Heap memory area.

Declaring Objects

When you create a class, you are creating a new data type. You can use this type to declare objects of that type. However, obtaining objects of a class is a **two-step process**.

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

Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new operator**.

Demo d1 = new demo( );

Object and Class Example: main within class

In this example, we have created a Student class that has two data members id and name. We are creating the object of the Student class by new keyword and printing the objects value.
Here, we are creating main() method inside the class.

Student.java

```
class Student
{
 int id;//field or data member or instance variable
 String name;
  public static void main(String args[])
  {
     Student s1=new Student();//creating an object of Student
     System.out.println(s1.id);//accessing member through reference variable
     System.out.println(s1.name);
 }
}
```

Output:

```
0
null
```

Object and Class Example: main outside class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different java files or single java file. If you define multiple classes in a single java source file, it is a good idea to save the file name with the class name which has main() method.

TestStudent1.java

```
class Student
 {
 int id;
 String name;
}
class TestStudent1{
 public static void main(String args[])
 {
  Student s1=new Student();
  System.out.println(s1.id);
  System.out.println(s1.name);
 }
}
```

Output:

```
0
null
```

Three Ways to initialize object

There are 3 ways to initialize object in java.

- By reference variable
- By method
- By constructor

1) Object and Class Example: Initialization through reference

Initializing object simply means storing data into object. Let's see a simple example where we are going to initialize object through reference variable.

TestStudent2.java

```
class Student{
int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Sonoo";
System.out.println(s1.id+" "+s1.name);//
printing members with a white space
 } }
```

Output:
101 Sonoo

We can also create multiple objects and store information in it through reference variable.

TestStudent3.java

```
class Student{
int id;
 String name; }
class TestStudent3{
 public static void main(String args[]) {
  Student s1=new Student();
  Student s2=new Student();
  s1.id=101;      //Initializing object
  s1.name="Sonoo";
  s2.id=102;      //Initializing object
  s2.name="Amit";
  System.out.println(s1.id+" "+s1.name);
  System.out.print(s2.id+" "+s2.name);
 } }
```

Output:
101 Sonoo
102 Amit

2) Object and Class Example: Initialization through method

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

TestStudent4.java

```
class Student{
 int rollno;
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;
 }
 void displayInformation(){System.out.pr
intln(rollno+" "+name);}
}
class TestStudent4{
 public static void main(String args[]) {
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"Karan");
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();
  s2.displayInformation();
 } }
```

Output:
111 Karan
222Aryan

As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

3) Object and Class Example: Initialization through constructor

We will learn about constructors in java later.

Arrays:

An array is a group of similar-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

One-Dimensional Arrays

A one-dimensional array is a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one dimensional array declaration is **type var-name[ ];** Here, type declares the base type of the array.

int month [ ];

Although this declaration establishes the fact that month is an array variable, no array actually exists. *In fact, the value of month is set to null, which represents an array with no value*.

To link *month with an actual, physical array of integers*, you must allocate one using *new* and assign it to month. **new is a special operator** that allocates memory.

The general form of new as it applies to one-dimensional arrays appears as follows:

array-var = new type[size];

Ex: month = new int[12];

After this statement executes, month will refer to an array of *12 integers*. Further, all elements in the array will be *initialized to zero*.

Another way to declare an array in single step is

type arr-name=new type[size];

Ex: int month = new int[12];

Arrays can be initialized when they are declared. For example, to store the number of days in each month, we do as follows

Example Program1: // An improved version of the previous program.

```
class AutoArray
{
  public static void main(String args[])
  {
   int month[ ] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,30, 31 };
  System.out.println("April has " + month[3] + " days.");
  }
}
```

When you run this program, in the output it prints the number of days in April. As mentioned, Java array indexes start with zero, so the number of days in **April has 30 days.**

Example Program2: // Average an array of values.

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

Output: Average is:12.3

Multidimensional Arrays

In Java, multidimensional arrays are actually **arrays of arrays**. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called ***twoD***.

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

Example Program:

```
// Demonstrate a two-dimensional array
class Testarray1
{
 public static void main(String args[])
 {

//declaring and initializing 2D array
int arr[][]={{1,2,3},{4,5,6},{7,8,9}};

 //printing 2D array
 for(int i=0;i<3;i++)
 {
   for(int j=0;j<3;j++)
  {
    System.out.print(arr[i][j]+" ");
   }
  System.out.println();
 }
}
}
```

Output:

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

Example Program:

```
//Demonstrate a three-dimensional array.
class threeDMatrix
{
   public static void main(String args[])
  {
int threeD[][][] = new int[3][4][5];
    int i, j, k;
    for(i=0; i<3; i++)
    for(j=0; j<4; j++)
    for(k=0; k<5; k++)
    threeD[i][j][k] = i * j * k;

    for(i=0; i<2; i++)
   {
    for(j=0; j<2; j++)
    {
      for(k=0; k<3; k++)
   System.out.print(threeD[i][j][k] + " ");
   System.out.println();
  }
 System.out.println();
} } }
```

Output:

```
0 0 0          0 0 0
0 0 0          0 1 2
```

We can create a **three-dimensional array** where first index specifies the number of tables, second one number o0f rows and the third number of columns.

Alternative Array Declaration Syntax
There is a second form that may be used to declare an array:
type[ ] var-name;

For example, the following two declarations are equivalent:
int al[ ] = new int[3];
int[ ] a2 = new int[3];

Constructors:

Constructor is a *special method of a class which is invoked automatically whenever an object is created*. It has same name that of a class. *A constructor initializes an object immediately upon creation.* A constructor has no return type; not even void. It cannot be abstract, final, native ,static or synchronized. **this** keyword refers another constructor in same class. A super keyword will call constructor of super class constructors are of two type

1.Default constructor
2.Parameterised constructor

1. Default constructor:
A constructor that accepts no parameters is called default constructor. If no constructor is defined for a class java system automatically generates the default constructor. **A default constructor is called when an instance is created for a class.** The default constructor automatically initializes all instance variables to zero.

Example:

```
class demo
{
int  x,y;
float  z;
demo()
{
x=1;
y=2;
z=3;
}
void display()
{
System.out.println("Values of x, y and z are:"+x+" "+y+" "+z);
}
}
class Demomain
{
public static void main(String args[])
{
demo d1=new demo( ); // this is a call for the above default constructor
d1.display();
}
}
```

Output: Values of x,y and z are: 1 2 3.0

Parameterized constructor: A constructor that takes arguments as parameters is called as parameterized constructor

```
class demo
{
int x;
int y;
float z;
 demo(int x1,int y1,int z1)
{
  x=x1;
y=y1;
z=z1;
}
void display()
{
System.out.println("Values of x, y and z are:"+x+" "+y+" "+z);
}
}
class Demomain1
{
public static void main(String args[])
{
demo d1=new demo(1,2,3); // this is a call for the above parameterized constructor
d1.display();
}
}
```

Output: Values of x,y and z are: 1 2 3.0

Java Copy Constructor

There is no copy constructor in java. But, we can copy the values of one object to another like copy constructor in C++. There are many ways to copy the values of one object into another in java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

In this example, we are going to copy the values of one object into another using java constructor.

Example:

```
class Student6{
   int id;
   String name;
   Student6(int i,String n){
   id = i;
   name = n;
   }
   Student6(Student6 s){
   id = s.id;
   name =s.name;
   }
  void display()
  {
  System.out.println(id+" "+name);  }
  public static void main(String args[])
  {
Student6 s1 = new Student6(111,"Karan");
Student6 s2 = new Student6(s1);
  s1.display();
  s2.display();
  } }
```

Output: 111 Karan
111 Karan

Methods:

Method is an action required by an object. Methods allow the programmer to modularize the program. All variables declared in method definitions are **local variables**. That means they are known only in the method in which they are defined. Most methods have a list of parameters that provide the means for communicating information between the methods. A methods parameters are also local variables.

There are several motivations for modularizing a program with methods.

- The divide and conquer approach makes program development more manageable.
- Another motivation is reusability. That means using existing methods as building blocks to create new programs.
- A third motivation is to avoid repeating code in a program.
- Packaging code as a method allows that code to be executed from several locations in a program simply by calling the method

Procedure:

• The only required elements of a method declaration are the ***method's return type, name, a pair of parentheses, (), and a body between braces, {}***.

• Two of the components of a method declaration comprise the **method signature** - ***the method's name and the parameter types***.

More generally, method declarations have six components, in order:

• **Modifiers:** such as public, private, and others you will learn about later.

• **The return type:** the data type of the value returned by the method, or void if the method does not return a value.

• **The method name:** the rules for field names apply to method names as well, but the convention is a little different.

• **The parameter list in parenthesis:** a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.

• **The method body, enclosed between braces:** the method's code, including the declaration of local variables, goes here.

Naming a Method

Although a method name can be any legal identifier, code conventions restrict method names. By convention, method names should be **a verb in lowercase or a multi-word name** that **begins with a verb in lowercase**, followed by adjectives, nouns, etc. In multi-word names, **the first letter of each of the second and following words should be capitalized**. Here are some **examples:**

run
runFast
getBackground
getFinalData

Typically, a method has a unique name within its class. However, a method might have the same name as other methods due to method overloading.

Parameter Passing:

Call by Value and Call by Reference in Java

There is only call by value in java, **not call by reference**. If we call a method passing a value, it is known as **call by value**. The changes being done in the called method, is not affected in the calling method.

Example of call by value in java

In case of call by value original value is not changed. Let's take a simple example:

```
class Operation{
 int data=50;
  void change(int data){
 data=data+100;//changes will be in the local
                                variable only
 }
  public static void main(String args[]){
   Operation op=new Operation();
System.out.println("before change "+op.data);
   op.change(500);
System.out.println("after change "+op.data);
  }
 }
```

Output: before change 50
after change 50

In case of call by reference original value is changed if we made changes in the called method. In this example ***we are passing object as a value***. **Example:**

```
 class Operation2{
  int data=50;
   void change(Operation2 op){
  op.data=op.data+100;//changes will be
                in the instance variable
  }
public static void main(String args[]){
   Operation2 op=new Operation2();
System.out.print("before:"+op.data);
   op.change(op);//passing object
System.out.println("after:"+op.data);
  } }
```

Output: before: 50
After: 150

Another Example: // Objects are passed by reference to achieve the concept of call-by-reference in JAVA

```
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// pass an object
void meth(Test o) {
  *=  2;
  /= 2;
}
}
class CallByRef {
public static void main(String args[]) {
Test ob = new Test(15, 20);
System.out.println("ob.a and ob.b before call: " +ob.a + " " + ob.b);
ob.meth(ob);
System.out.println("ob.a and ob.b after call: " +ob.a + " " + ob.b);
}     }
```

Output:

ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10

static : (constant / fixed)

It can be applied to data members, methods and some inner classes.

static data members :

1. static data members can be accessed by only static methods as well as non static.
2. static data members are not instance – specific. They are common to all objects of a class. Therefore, they are known as class variables.
3. They are not destroyed between function calls.
4. They can be accessed directly with the class name. **E.g., classname.variable_name;**
5. For all objects of a class, only one copy of static members exists in memory and it is shared by all objects. It is contrary to instance variables.
6. static variables are declared as **static int x ;**

static methods :

1. static method is a method whose definition is preceded by keyword **static**.
2. static methods can act upon only static variables.
3. static method can be **called directly using class name without creating object**.
4. The keyword **"this" can" t be used** inside a static method, as static members don‘ t belongs to any particular object.
5. **A static method can call only other static methods .**

static block:

1. It is a piece of code enclosed in braces preceded by the keyword **static**.
2. static block is executed even **before main()** .
3. It is used to initialize some variables or any such things.
4. **It can use only static variables & call only static methods**.

The following example shows a class that has a static method, some static variables, and a static initialization block:

// Demonstrate static variables, methods, and blocks: UseStatic.java

```
class UseStatic {
static int a = 3;      static int b;
static void math(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {          //Static block
System.out.println("Staticblock initialized.");
b = a * 4;
}
public static void main(String args[]) {
math(42);
} }
```

Syntax:

```
static
{
  statements;
----------
 }
```

Output:

x=42

a=3

b=12

To call a static method from outside its class, use following general form:

classname.method( )

Here is an example. Inside main( ), the static method callme( ) and the static variable b are accessed through their class name StaticDemo.

Ex: StaticByName.java

```
class StaticDemo {
static int a = 42;
static int b = 99;
static void callme() {
System.out.println("a = " + a);
}
}
class StaticByName {
public static void main(String args[]) {
StaticDemo.callme();
System.out.println("b = " + StaticDemo.b);
}
}
```

Output:

a = 42
b = 99

Concrete class: (real / actual)

Any class that can be instantiated is known as **concrete class**. All **non-abstract class** is nothing but concrete classes.

Sno	Constructor	Method
1	Creates an instance of a class	Group of statements
2	Cannot be abstract ,final,static,native or synchronized	Can be final , abstract,static, native or synchronized
3	NO return type not even void	Will have all possible return types including void
4	Has same that of a class	Can be given any valid name
5	this keyword Refers another constructor in the same class.	this keyword refers to an instance of the class . cannot be used with static methods
6	Super keyword is used ot call constructor of super class	super calls the super class overridden method
7	By default no arguments are supplied to constructor	No default supply of arguments

Sno	Instance method	Static method
1	Instance methods perform an operation on individual objects of a class	Static methods perform an operation for entire class
2	Instance methods uses instance variable of that object	Static method do not use instance variable of object of the class
3	Instance method in invoked by an object variable followed by dot operator followed by method name	Static method in invoked by Class name followed by dot operator followed by method name
4	Instance method cannot be invoked unless an instance of the class has already created	Static methods can be invoked without creating an instance of the class

Access Specifiers:

There are 4 types of java access modifiers: *private, public, protected and default*

Access Specifiers are the **keywords** that alter the meaning and accessibility (scope) of a data member of a class, method or constructor. They are, private, public, protected and default. There are many non-access modifiers such as static, final, abstract, native, synchronized, transient and volatile.

private:

It can be applied to variables and methods. Private members of a class are **accessible from within the class only**. They cannot be accessed from outside the class and its subclass.

Ex: class A

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

Output: compile time error

Role of Private Constructor: If you make any class constructor private, you cannot create the instance of that class from outside the class.

Ex: **class** A{

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

Output: compile time error

Note: A class cannot be private or protected except nested class.

public:

It can be applied to variables (data members), methods and classes. Public member of class can be **accessed globally** any other code. They can be accessed from outside the class and sub class. A public class can be **accessed from any package**.

```
class Alpha
{
public int i;
```

```
    public void publicMethod()
    {
       System.out.println("in PublicMethod");
       System.out.println("i="+i);
     }
}
class Beta
{
  public static void main(String args[])
  {
    Alpha a=new Alpha();
     a.i=10;
     a.publicMethod();
  }
}
```

Output:

in PublicMethod
i=10

protected:

It can be applied to data members and methods. Protected member of a class can be accessed from **within the class**, **sub class in the same package** and **sub class in different package**. It cannot be accessed from a non-subclass in different package.

default: When **no access specifier** is mentioned for a data member, method or class. It assumes default / friend. There are **no keywords such as default or friend**. The default / friend members of a class can be accessed from anywhere except from sub class in different package and non-sub class from different package. (**Ex:** Public access specifier program without public)

	Variable	Method	Class	within class	within package	outside package by subclass only	outside package
private	Y	Y	N	Y	N	N	N
public	Y	Y	Y	Y	Y	Y	Y
protected	Y	Y	N	Y	Y	Y	N
default	Y	Y	Y	Y	Y	N	N
static	Y	Y	Y	-	-	-	-
final	Y	Y	Y	-	-	-	-

this Keyword:

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this keyword**. **this** can be used inside any method to refer to the current object. That is, this is always a reference to the object on which the method was invoked.

For example, if you have an Employee class with members 'firstName' and 'lastName' and let's say you have a private method that formats a full name everytime there is a change in either the first name or the last name. Then the formatFullName method can access the mebers using 'this' keyword as follows:

```
private void formatFullName()
{
        this.fullName = String.format("%s %s", this.firstName, this.lastName);
}
```

Example 1:

```
class demo
{
int x;
int y;
float z;
 demo(int x,int y,int z)
{
  this.x=x;
  this.y=y;
  this.z=z;
}
void display()
{
System.out.println("Values of x, y and z are:"+x+" "+y+" "+z);
}
}
class Demomain2
{
public static void main(String args[])
{
demo d1=new demo(1,2,3); // this is a call for the above parameterized constructor
d1.display();
} }
```

Output: Values of x, y and z are:1 2 3.0

Example 2:

```
class ThisRef
{
   int age=40;
   ThisRef(int age)
   {
     this.age = age;
     System.out.println(age);
   }
        public static void main(String args[])
        {
          ThisRef s=new ThisRef(20);
          System.out.print(s.age);
        }
}
```

Op: 20
20 //without using this keyword 40

To differentiate between the local and instance variables we have used this keyword in the constructor.

Garbage Collection

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects. To do so, we were using **free() function in C language** and **delete() in C++.** But, in **java** it is performed **automatically**. So, java provides better memory management.

Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM).

How can an object be unreferenced?

Object can be unreferenced using the following ways:

- By nulling the reference
- By assigning a reference to another
- By annonymous object

1) By nulling a reference:

```
Employee e=new Employee();
e=null;
```

2) By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collection
```

3) By annonymous object:

```
new Employee();
```

finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

```
protected void finalize(){}
```

Note: *The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).*

gc( ) method

The gc( ) method is used to invoke the garbage collector to perform cleanup processing. The **gc()** is found in System and Runtime classes. **public static void** gc( ){}

Simple Example of garbage collection in java

```
public class TestGarbage1{
 public void finalize(){System.out.println("object is garbage collected");}
 public static void main(String args[]){
  TestGarbage1 s1=new TestGarbage1();
TestGarbage1 s2=new TestGarbage1();
  s1=null;
  s2=null;
  System.gc();
 } }
```

Output:

```
object is garbage collected
object is garbage collected
```

Note: ***Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.***

Method Overloading in Java

If a class has 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 we 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 behaviour of the method because **its name differs**.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

Note: In java, Method Overloading is not possible by changing the return type of the method.

In this example, we have created **two overloaded methods**, first sum method performs **addition of two numbers** and second sum method performs **addition of three numbers**.

```
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 Calculation();
  obj.sum(10,10,10);
  obj.sum(20,20);
  }
} Output: 30
          40
```

In this example, we have created **two overloaded methods that differ in data type**. The first sum method receives **two integer arguments** and second sum method receives **two double arguments**.

```
 class Calculation2
 {
   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[]){
  Calculation2 obj=new Calculation2();
  obj.sum(10.5,10.5);
  obj.sum(20,20);
    }   }   Output: 21.0
                    40
```

In java, method overloading is not possible by changing the return type of the method because there may occur ambiguity. Let's see how ambiguity may occur:

```
 class Calculation3{
    int sum(int a,int b){System.out.println(a+b);}
   double sum(int a,int b){System.out.println(a+b);}
   public static void main(String args[]){
   Calculation3 obj=new Calculation3();
   int result=obj.sum(20,20); //Compile Time Error
   } }
```

result=obj.sum(20,20); //Here how can java determine which sum() method should be called.

We can overload main() method in JAVA by using method overloading. You can have any number of main methods in a class by method overloading. Let's see the simple example:

```
 class Overloading1{
   public static void main(int a)
   {
   System.out.println(a);
   }
   public static void main(String args[ ])
   {
System.out.println("main() method invoked");
 main(10);
   }
 }
```

Output: main() method invoked
10

Constructor Overloading in Java

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 by taking into account the number of parameters in the list and their type.

Example of Constructor Overloading

```
class Student5{
   int id;
   String name;
   int age;
   Student5(int i,String n){
   id = i;
   name = n;
   }
   Student5(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[ ])
   {
Student5 s1 = new Student5(11,"MRCET");
Student5 s2 = new Student5(22,"CSE",25);
   s1.display( );
   s2.display( );
  }
}
```

Output:

```
 11 MRCET 0
 22 CSE 25
```

Recursion:

Java supports recursion. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself.

A method that calls itself is said to be recursive.

Example: Write a program to display Fibonacci series without using Recursion.

```
public class Fib
{
public static void main(String args[]){
          String n1;
          int f1=0,f2=1,n,i,f3;
          n1="5";
          n=Integer.parseInt(n1);
        System.out.println("Series\n"+f1+"\n"+f2);
        for(i=1;i<n;i++)
          {
            f3=f1+f2;
          System.out.println(f3);
          f1=f2;
            f2=f3;
          }
       }
}
```

Output: Series: 0 1 1 2 3 5

// recursive declaration of method fibonacci

```
public class MainClass1 {
 public static long fib(long no)
{
    if ((no == 0) || (no == 1))
       return no;
    else
      // recursion step
      return fib(no - 1) + fib(no - 2);
  }
  public static void main(String[] args)
{
 System.out.printf("Fibonacci of 10:\n");
   for (int coun = 0; coun < 10; coun++)
   System.out.printf("%d\t", fib(coun));
  }
}
```

Output: Fibonacci of 10:

```
0   1   1   2   3   5    8 13 21 34
```

Example: Factorial of a number using Recursion.
The factorial of a number N is the product of all the whole numbers between 1 and N. For example, 3 factorial is 1 × 2 × 3, or 6.

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

Output:

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

BufferedReader class (or) Console Input and Output:

In Java, **console input** is accomplished by reading from **System.in** or **reading data from keyboard**. To obtain a character-based stream that is attached to the console, wrap **System.in** in a **BufferedReader** object. **BufferedReader** supports a buffered input stream. Reader is an abstract class. One of its concrete subclasses is **InputStreamReader**, which converts bytes to characters.

To obtain an **InputStreamReader** object that is linked to **System.in**, the following line of code creates **a BufferedReader that is connected to the keyboard:**

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

After this statement executes, br is a character-based stream that is linked to the console through **System.in**.

Reading Characters

To read a character from a BufferedReader, use read( ). The version of read( ) that we will be using is **int read( ) throws IOException**. Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value. It returns – 1 when the end of the stream is encountered. As you can see, it can throw an IOException.

Example Program: // Use a BufferedReader to read characters from the console.

```
import java.io.*;
class BRRead {
public static void main(String args[]) throws IOException
{
char c;
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter characters, 'q' to quit.");
do {                    // read characters
c = (char) br.read();
System.out.println(c);
} while(c != 'q');
}
}
```

Output:

```
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q
```

Note: This output may look a little different from what you expected, because System.in is line buffered, by default. This means that no input is actually passed to the program until you **press ENTER**.

Reading Strings

To read a string from the keyboard, use the version of readLine( ) that is a member of the BufferedReader class. Its general form is shown here:

String readLine( ) throws IOException

The following program demonstrates BufferedReader and the readLine() method;
The program reads and displays lines of text until you enter the word –stop|:

Example Program: // Read a string from console using a BufferedReader

```
import java.io.*;
class BRReadLines {
public static void main(String args[])
throws IOException
{
// create a BufferedReader using System.in
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String str;
System.out.println("Enter lines of text.");
System.out.println("Enter 'stop' to quit.");
do {
str = br.readLine();
System.out.println(str);
} while(!str.equals("stop"));
}
}
```

Java Scanner class

There are various ways to **read input from the keyboard**, the **java.util.Scanner class** is one of them. The **Java Scanner** class breaks the input into tokens using a delimiter that is whitespace bydefault. It provides many methods to read and parse various primitive values.

Java Scanner class is widely used to parse text for string and primitive types using regular expression.

Java Scanner class extends Object class and implements Iterator and Closeable interfaces.

There is a list of commonly used Scanner class methods:

Method	Description
public String next()	it returns the next token from the scanner.
public String nextLine()	it moves the scanner position to the next line and returns the value as a string.
public byte nextByte()	it scans the next token as a byte.
public short nextShort()	it scans the next token as a short value.
public int nextInt()	it scans the next token as an int value.
public double nextDouble()	it scans the next token as a double value.
public float nextDouble()	it scans the next token as a float value.

Java Scanner Example to get input from console

Simple example of the Java Scanner class which reads the ***int, string & double*** value as an input:

```
import java.util.Scanner;
 class ScannerTest
 {
  public static void main(String args[])
   {
    Scanner sc=new Scanner(System.in);
    System.out.println("Enter your rollno");
    int rollno=sc.nextInt();
    System.out.println("Enter your name");
    String name=sc.next();
    System.out.println("Enter your fee");
    double fee=sc.nextDouble();
    System.out.println("Rollno:"+rollno+
        " name:"+name+" fee:"+fee);
    sc.close();
   }
 }
```

Output:

```
Enter your rollno
111
Enter your name
Kalpana
Enter
450000
Rollno:111 name:Kalpana fee:450000
```

Strings:

Java String provides a lot of concepts that can be performed on a string such as compare, concat, equals, split, length, replace, compareTo, intern, substring etc.

In java, string is basically an object that represents sequence of char values. An array of characters works same as java string. For example:

```
char[] ch={'j','a','v','a','t','p','o','i','n','t'};
String s=new String(ch);
```

Above code is same as: **String s="javatpoint";**

The java String is **immutable** i.e. it cannot be changed but a new instance is created. For **mutable class**, you can use ***StringBuffer and StringBuilder class***.

What is String in java

Generally, string is a sequence of characters. But in java, ***string is an object*** that represents a sequence of characters. String class is used to create string object.

There are two ways to create String object:

1. By string literal
2. By new keyword

1) String Literal

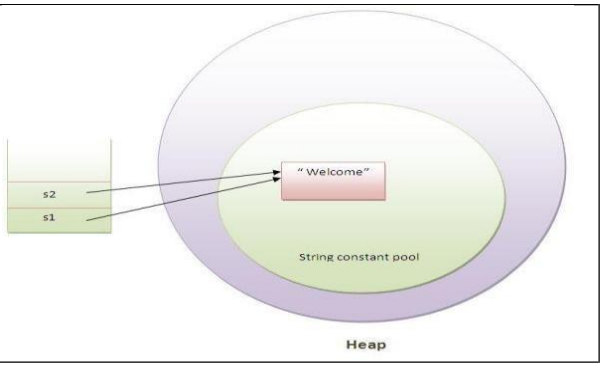
Java String literal is created by using double quotes.

For Example:

```
String s="welcome";
```

Each time you create a string literal, the **JVM checks** the **string constant pool** first. **If the string already exists in the pool, a reference to the pooled instance is returned**. If string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
String s1="Welcome";
String s2="Welcome"; //will not create new instance
```

In the beside example only one object will be created. Firstly JVM will not find any string object with the value "Welcome" in string constant pool, so it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create new object but will return the reference to the same instance.	

Note: String objects are stored in a special memory area known as string constant pool.

Java has more memory efficient, because no new objects are created if it exists already in string constant pool.

2) By new keyword

String s=**new** String("Welcome"); //creates two objects and one reference variable

Java String Example

```
public class StringExample{
public static void main(String args[]){
String s1="java";//creating string by java string literal
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch); //converting char array to string
String s3=new String("example"); //creating java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

Output:
java
strings
example

Difference between StringBuffer and StringBuilder

There are many differences between StringBuffer and StringBuilder. A list of differences between StringBuffer and StringBuilder are given below:

No.	StringBuffer	StringBuilder
1)	StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
2)	StringBuffer is *less efficient* than StringBuilder.	StringBuilder is *more efficient* than StringBuffer.

StringBuffer Example

```
public class BufferTest{
public static void main(String[] args) {
StringBuffer buffer=new StringBuffer("hello");
        buffer.append("java");
       System.out.println(buffer);
   }
}
```

OP: hellojava

StringBuilder Example

```
public class BuilderTest{
public static void main(String[] args) {
StringBuilder builder=new StringBuilder("hello");
        builder.append("java");
       System.out.println(builder);
   }
}
```

OP: hellojava

Immutable String in Java

In java, **string objects are immutable**. Immutable simply means ***unmodifiable*** or ***unchangeable***. Let's try to understand the immutability concept by the example given below:

```
class Testimmutablestring{
 public static void main(String args[]){
   String s="Sachin";
   s.concat(" Tendulkar"); //concat() method appends the string at the end
   System.out.println(s);     //will print Sachin because strings are immutable objects
 } }
```

Output: Sachin

String Handling Methods:

String compare by compareTo( ) method

The String compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.
Suppose s1 and s2 are two string variables. If:

s1 == s2 :0 **s1 > s2** :positive value **s1 < s2** :negative value

String compare by == *operator*

The = = operator compares references not values.

```
class Teststringcomparison3
{
 public static void main(String args[ ])
{
   String s1="Sachin";
   String s2="Sachin";
   String s3=new String("Sachin");
   System.out.println(s1==s2);//true (because
          both refer to same instance)
   System.out.println(s1==s3);//false(because s
       3 refers to instance created in nonpool)
 }
}
```

Output: true
false

String compare by *compareTo method*

```
class Teststringcomparison4
{
public static void main(String args[])
{
   String s1=" kalpana";
   String s2=" kalpana";
   String s3="Mrcet";
System.out.println(s1.compareTo(s2));//0
System.out.println(s1.compareTo(s3));
                    //1(because s1>s3)
System.out.println(s3.compareTo(s1));
                    //-1(because s3 < s1 )
 System.out.println(s1.equals(s2));//true
  }
 }
```

Output: 0 1 -1 true

String Concatenation by + (string concatenation) operator

In java, string concatenation forms a new string *that is* the combination of multiple strings. There are two ways to concat string in java:

1. By + (string concatenation) operator
2. By concat() method

Java string concatenation operator (+) is used to add strings. **For Example:**

```
class TestStringConcatenation1{
 public static void main(String args[]){
   String s="Kalpana"+" Java";
   System.out.println(s); // Kalpana Java
 }
}
```

Output: Sachin Tendulkar

For example concat():

```
class Testimmutablestring1{
 public static void main(String args[]){
   String s="Kalpana";
   s=s.concat(" Java");
   System.out.println(s);
 }
}
```

Output: Kalpana Java

Substring in Java

A part of string is called **substring**.

In case of string:

- **startIndex:** inclusive
- **endIndex:** exclusive

Let's understand the startIndex and endIndex by the code given below.

```
String s="hello";
System.out.println(s.substring(0,2));//he
```

In the above substring, 0 points to h but 2 points to e (because end index is exclusive).

Example of java substring

```
public class TestSubstring{
 public static void main(String args[]){
   String s="Sachin Tendulkar";
   System.out.println(s.substring(6));//Tendulkar
   System.out.println(s.substring(0,6));//Sachin
 }
}
```

Output:

Tendulkar
Sachin

Java String toUpperCase() and toLowerCase() method

The java string toUpperCase() method converts this string into uppercase letter and string toLowerCase() method into lowercase letter.

```
String s="Kalpana";
System.out.println(s.toUpperCase());//KALPANA
System.out.println(s.toLowerCase());//kalpana
System.out.println(s); // Kalpana (no change in original)
```

Output:
KALPANA
kalpana
Kalpana

Java String trim() method

The string trim() method **eliminates white spaces** before and after string.

```
String s=" Kalpana  ";

System.out.println(s);          //  Kalpana
System.out.println(s.trim()); //Kalpana
```

Output:
Kalpana
Kalpana

Java String startsWith() and endsWith() method

```
String s="Kalpana";

System.out.println(s.startsWith("Ka"));  //true
System.out.println(s.endsWith("a"));  //true
```

Output:
true
true

Java String charAt() method

The string charAt() method returns a character at specified index.

```
String s="Sachin";
System.out.println(s.charAt(0));//S
System.out.println(s.charAt(3));//h
```

Output:
S
h

Java String length() method

The string length() method returns length of the string.

```
String s="Kalpana ";
System.out.println(s.length());//7
```

Output:
7

Java String replace() method

The string replace() method replaces all occurrence of first sequence of character with second sequence of character.

```
String s1="Java is a programming language. Java supports OOP features.";
String replaceString=s1.replace("Java","CPP"); //replaces all occurrences of "Java" to "CPP"
System.out.println(replaceString);
```

Output: CPP is a programming language. CPP supports OOP features.

StringTokenizer in Java

The java.util.StringTokenizer class allows you to **break a string into tokens**. It is simple way to break string.

```
import java.util.StringTokenizer;
public class Simple
 {
   public static void main(String args[])
   {
     StringTokenizer st = new StringTokenizer("my name is kalpana"," ");
      while (st.hasMoreTokens())
        {
        System.out.println(st.nextToken());
        }
    }
   }
```

Output:

```
my
name
is
kalpana
```

boolean hasMoreTokens( ) : checks if there is more tokens available.

String nextToken( ) : returns the next token from the StringTokenizer object.

StringTokenizer(String str) : creates StringTokenizer with specified string.

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	**Syntax:**

class Java_Outer_class{
//code
class Java_Inner_class{
//code
}
}
	class Outer{
	//code
	class Inner{
	//code
	} } //object creation of inner class ⇩
	Outer.Inner obj=Outerobject.new Inner();

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.

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 TestOuter
{
  private int data=30;
   class Inner
    {
      void msg()
      {
        System.out.println("data is "+data);
      }
    }
   public static void main(String args[])
   {
     TestOuter obj=new TestOuter();
     TestOuter.Inner in=obj.new Inner();
     in.msg();
   }
}
```

Output:

data is 30

Internal working of Java member inner class

The java compiler creates two class files in case of inner class. The class file name of inner class is **"Outer$Inner"**.

UNIT II - Inheritance – Types of Inheritance, Member access rules, super keyword, and preventing inheritance: final classes and methods.

Polymorphism – dynamic binding, method overriding, abstract classes and methods.

Interfaces- Interfaces Vs Abstract classes, defining an interface, implement interfaces, accessing implementations through interface references, extending interface.

Packages- Defining, creating and accessing a package, Understanding CLASSPATH, importing packages.

Inheritance

Inheritance can be defined as the process where **one class acquires the properties methods and fields of another**. With the use of inheritance the information is made manageable in a **hierarchical order**. The class which inherits the properties of other is known as **subclass derived class, child class** and the class whose properties are inherited is known as **super class base class, parent class.**

Inheritance is the mechanism of **deriving new class from old one**, **old class is known as super class** and **new class is known as subclass**. The subclass inherits all of its instances variables and methods defined by the super class and it also adds its own unique elements. Thus we can say that subclass is specialized version of super class.

Benefits of Java's Inheritance

1. Reusability of code
2. Code Sharing
3. Consistency in using an interface

Classes

Superclass(Base Class)	Subclass(Child Class)
It is a class from which other classes can be derived.	It is a class that inherits some or all members from superclass.

extends Keyword

extends is the **keyword** used to inherit the properties of a class. Below given is the syntax of extends keyword.

```
class Super{
..... ..... }
class Sub extends Super{
..... ..... }
```

Base Class

Derived Class

Types of Inheritance in Java

There are various types of inheritance as demonstrated below

Single Level Inheritance:
When one base class is being inherited by one sub class then that kind of inheritance is known as single level inheritance.

Multi Level Inheritance:
When a sub class is in turn being inherited then that kind of inheritance is known as multi level inheritance.

Hierarchical Inheritance:
When a base class is being inherited by one or more sub class then that kind of inheritance is known as hierarchical inheritance.

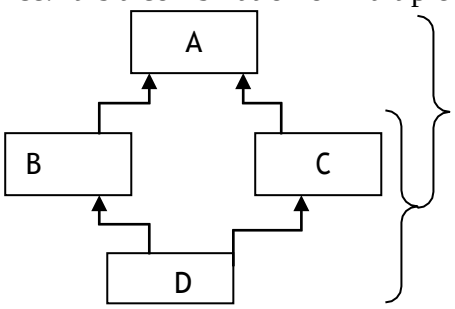
Multiple inheritance:

Deriving one subclass from more than one super classes is called multiple inheritance.

Eg. class A { statements ; }
class B { statements ; }
class C extends A, B { statements ; }

Note: Java does not support multiple inheritance with classes. But, it supports multiple inheritance using interfaces.

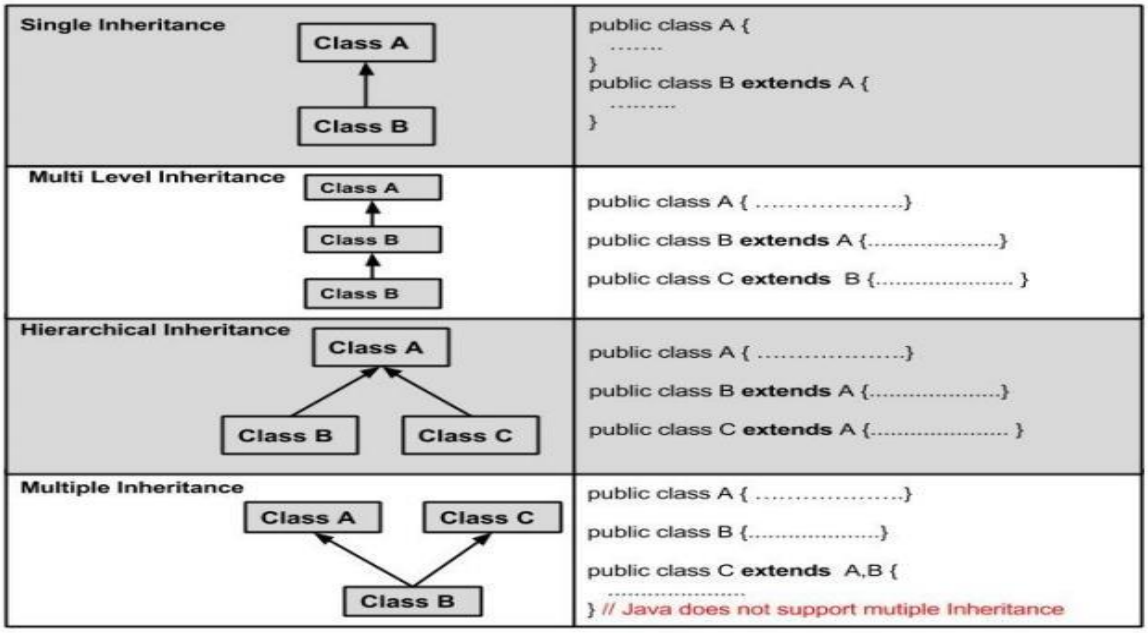
Hybrid Inheritance: It is a combination of multiple and hierarchical inheritance.



HIERARCHICAL INHERITANCE

MULTIPLE INHERITANCE

A sub class uses the keyword **"extends"** to inherit a base class.



Example for Single Inheritance:

```
class x
{
        int a;
       void display()     {
            a=0;
           System.out.println(a);
      } }
  class y extends x {
          int b;
           void show( )
         {
              B=1;
              System.out.println(b);
       } }
class show_main
{
        public static void main(String args[])
        {
            y y1=new y();
            y1.display();
            y1.show();
        } }
```

Output:

0
1

Since the class y is inheriting class x, it is able to access the members of class x.
Hence the method display() can be invoked by the instance of the class y.

Example for multilevel inheritance:

```
class x
{
int a;
      void display()
      {
            a=0;
           System.out.println(a);
       }
}
class y extends x
{
        int b;
       void show()
       {
             B=1;
            System.out.println(b);
        }
}
class z extends y
{
    int c;
    show1()
    {
          c=2;
         System.out.println(c);
    }
}
   class show_main
   {
      public static void main(String args[])
      {
            z z1=new z();
             z1.display();
         z1.show();
      }
}
```

Output:

```
0
1
2
```

Since class z is inheriting class y which is in turn a sub class of the class x, indirectly z can access the members of class x. Hence the instance of class z can access the display () method in class x, the show () method in class y.

Problems:

In java multiple level inheritance is not possible easily. We have to make use of **a concept called interfaces to achieve it.**

Access Specifers:

Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages.

Classes act as containers for data and code. We can protect the data from unauthorized access. To do this ,we are using access specifiers. An access specifier is a keyword that is used to specify how to access a member of a class or the class itself. There are four access specifiers in java:

- Public
- Private
- Protected
- Default
- Privateprotected

1. Private members can be accessed only within the class in which they are declared.
2. Protected members can be accessed inside the class in which they are declared and also in the sub classes in the same package and sub classes in the other packages.
3. Default members are accessed by the methods in their own class and in the sub classes of the same package.
4. Public members can be accessed anywhere.

Example of Member Access and Inheritance

Case 1: Member Variables have **no access modifier** (**default** access modifier)

```
class inherit1
{
    int i=10;
    void meth0()
    {
        System.out.println("Value of i in Base Class is:"+i);
    }
}
class inherit2 extends inherit1
{
    void meth1()
    {
        System.out.println("Value of i in Child Class is:"+(i*5));
    }
}
class inherittest
{
    public static void main(String args[])
    {
        inherit2 ob1 = new inherit2();
        ob1.meth1();
    }
}
```

Variable "i" with no access modifier and we are using it in child class. Program runs with no error in this case as members with default access modifier can be used in child class

Output

```
C:\Achin Jain>javac inherit2.java

C:\Achin Jain>java inherittest
Value of i in Child Class is:50
```

Case2: Member Variables have public/protected access modifier. If a member of class is declared as **either public or protected** than it can be accessed from **child or inherited class**.

```
class inherit1
{
    public int i=10;
    protected int j=5;
    void meth0()
    {
        System.out.println("Value of i in Base Class is:"+i);
    }
}
class inherit2 extends inherit1
{
    void meth1()
    {
        j=j+i;
        System.out.println("Value of j in Child Class is:"+j);
    }
}
class inherittest
{
    public static void main(String args[])
    {
        inherit2 ob1 = new inherit2();
        ob1.meth1();
    }
}
```

Output: Value of j in Child Class is: 6

Case3: Member Variables have private access modifier. If a member of class is declared as private than it cannot be accessed outside the class not even in the inherited class.

```
class inherit1
{
    public int i=10;
    private int j=5;
    void meth0()
    {
        System.out.println("Value of j in Base Class is:"+i);
    }
}
class inherit2 extends inherit1
{
    void meth1()
    {
        j=j+i;
        System.out.println("Value of j in Child Class is:"+j);
    }
}
```

Variable „j" is declared as Private in class inherit1 which means it cannot be accessed outside the class scope.

In the inherited class an attempt to modify the value of a private variable is made which results in compile time error

Output:

```
C:\Achin Jain>javac inherit2.java
inherit2.java:14: error: j has private access in inherit1
                j=j+i;

inherit2.java:14: error: j has private access in inherit1
                j=j+i;

inherit2.java:15: error: j has private access in inherit1
                System.out.println("Value of j in Child Class is:"+j);

3 errors
```

Why multiple inheritance is not supported in java?

When a class extends multiple classes is known as multiple inheritance. To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.

Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error now.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were
 public Static void main(String args[]){
 C obj=new C();
   obj.msg();//Now which msg() method would be invoked?
}
}
```

Output: Compile Time Error

Note: Multiple inheritance is not supported in java through class.

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Super Keyword:

Whenever a sub class needs to refer to its immediate super class, we can use the super keyword. **Super has two general forms**

- The first calls the super class constructor.
- The second is used to access a member of the super class that has been hidden by a member of a sub class

Syntax:

A sub class can call a constructor defined by its super class by use of the following form of super.

super(arg-list);

here arg-list specifies any arguments needed by the constructor in the super class .

The **second form** of super acts like a**"this" keyword**. The difference between "this" and "super" is that **"this" is used to refer the current object** where as the **super is used to refer to the super class.**

The usage has the following general form:

super.member;

Example:

```
class x
{
      int a;
     x( )
     {
       a=0;
     }
    void display( )
    {
       System.out.println(a);
     }
}
class y extends x
{
       int b;
     y( )
    {
      super( );
       b=1;
     }
void display( )
    {
       super.display( );
       System.out.println(b);
     }
}
class super_main
{
     public static void main(String args[ ])
     {
         y y1=new y( );
         y1.display( );
      }
}
```

Output: 0 1

super is a reference variable that is used to **refer immediate parent class object**. Uses of super keyword are as follows:

1. super() is used to invoke immediate parent class constructors
2. super is used to invoke immediate parent class method
3. super is used to refer immediate parent class variable

Example:

```
class inherit1
{
    int i=10;
}
class inherit2 extends inherit1
{
    int i=20;
    void meth1()
    {
        System.out.println("Value of Parent class variable i is :"+super.i);
        System.out.println("Value of Child class variable i is :"+i);
    }
}
class inherittest
{
    public static void main(String args[])
    {
        inherit2 ob1 = new inherit2();
        ob1.meth1();
    }
}
```

To print the value of Base class variable in a child class use ***super.variable*** name

This "i" will print Value of local class variable

Output:

```
C:\Achin Jain>javac inherit2.java

C:\Achin Jain>java inherittest
Value of Parent class variable i is :10
Value of Child class variable i is :20
```

Example to call Immediate Parent Class Constructor using super Keyword

The super() keyword can be used to invoke the Parent class constructor as shown in the example below.

```
class inherit1
{
    inherit1()
    {
        int i=10;
        System.out.println("Base Class COnstructor is invoked");
    }
}
class inherit2 extends inherit1
{
    int i=20;
    inherit2()
    {
        super();
    }
}
class inherittest
{
    public static void main(String args[])
    {
        inherit2 ob1 = new inherit2();
    }
}
```

Output:

```
C:\Achin Jain>javac inherit2.java

C:\Achin Jain>java inherittest
Base Class COnstructor is invoked
```

Note: super() is added in each class constructor automatically by compiler. As default constructor is provided by compiler automatically but it also adds super() for the first statement. If you are creating your own constructor and you don‟t have super() as the first statement, compiler will provide super() as the first statement of the constructor.

Example where super() is provided by compiler

```
class inherit2 extends inherit1
{
    int i=20;
    inherit2()
    {
    }
}
```

Same program as above but no super keyword is used. You can see in the o/p below that compiler implicitly adds the super() keyword and same output is seen.

Output:

```
C:\Achin Jain>javac inherit2.java

C:\Achin Jain>java inherittest
Base Class COnstructor is invoked
```

Preventing Inheritance:

Final Keyword: (last / concluding)

Final keyword can be used in the following ways:

Final Variable: Once a variable is declared as final, its value cannot be changed during the scope of the program
Final Method: Method declared as final cannot be overridden
Final Class: A final class cannot be inherited

Using final with inheritance:
The keyword **final** has three uses.

- First it can be used to create the equivalent of a **named constant**.
- To prevent overriding.
- To prevent inheritance.

final data members:
Data member with final modifier becomes a constant.

Using final to prevent overriding:
To disallow a method from being overridden, specify final as a modifier at the start of the declaration.

Methods declared as final cannot be overridden.

Syntax: final <return type> <method name> (argument list);

```
final returntype funname()
{ - - - - -
  }
```

Using final to prevent inheritance:

Some times we may want to **prevent a class from being inherited**. In order to do this we must precede the class declaration **with final keyword**.

Declaring a class as final implicitly declares all its methods as final. **The final class can' t be sub classed or derived**. This means that we can‟ t create a sub class from a final class.

Example:
final class A { class
B extends A{
} }

The above code gives compile error. Because class A is a final class, which can‟ t be derived.

Example of Final Variable: Final variables work like const of C-language that can‟t be altered in the whole program. That is, final variables once created can‟t be changed and they must be used as it is by all the program code.

```
class finalvar
{
    final int i=10;
    finalvar()
    {
        System.out.println("Value of Final Variable i is :"+i);
        i=i+10;
        System.out.println("Value of Final Variable i after change is :"+i);
    }
}
class finaltest
{
    public static void main(String args[])
    {
        finalvar obj1 = new finalvar();
    }
}
```

Variable “i” is declared as final

In this code we are trying to modify the value of a Final Variable which will result in error as shown in the output.

Output:

```
C:\Achin Jain>javac finaltest.java
finaltest.java:7: error: cannot assign a value to final variable i
                i=i+10;
                ^
1 error
```

Example of Final Method:

Generally, a super class method can be overridden by the subclass if it wants a different functionality. If the super class desires that the subclass should not override its method, it declares the method as final. (it gives compilation error).

```
class finalmethod
{
    int i=10;
    final void meth1()
    {
        System.out.println("Value of Final Variable i is :"+i);
    }
}
class childclass extends finalmethod
{
    final void meth1()
    {
        System.out.println("Value of Final Variable i is :"+i);
    }
}
class finaltest
{
    public static void main(String args[])
    {
        childclass obj1 = new childclass();
        obj1.meth1();
    }
}
```

Output

```
C:\Achin Jain>javac finaltest.java
finaltest.java:11: error: meth1() in childclass cannot override meth1() in final
method
        final void meth1()
                   ^
  overridden method is final
1 error
```

Example of Final Class: If we want the class not be sub-classed(or extended) by any other class, declare it final. Classes declared final can not be extended.

Output

Polymorphism

Polymorphism came from the two Greek words „**poly**" means many and „**morphos**" means forms. If the same method has ability to take more than one form to perform several tasks then it is called **polymorphism**.

It is of two types: Dynamic polymorphism and Static polymorphism.

Dynamic Polymorphism / Dynamic binding:

The polymorphism exhibited **at run time** is called **dynamic polymorphism**. In this dynamic polymorphism a method call is **linked with method body at the time of execution by JVM**. Java compiler does not know which method is called at the time of compilation. This is also known as **dynamic binding** or **run time polymorphism.**

Method overloading and method overriding are examples of Dynamic Polymorphism in Java.

o Method Overloading: Writing two or more methods with the **same name**, but with a **difference in the method signatures** is called **method over loading**.

Method signature represents the method name along with the method parameters. Method is called depending upon the difference in the method signature. The difference may be due to the following:

There is a difference in the no. of parameters.

```
void add (int a,int b)
void add (int a,int b,int c)
```

difference in the data types of parameters.

```
void add (int a,float b)
void add (double a,double b)
```

difference in the sequence of parameters.

```
void swap (int a,char b)
void swap (char a,int b)
```

//Dynamic polymorphism: Overloading methods

```
class Sample
{ void add(int a,int b)
   { s
 System.out.println ("sum of two="+ (a+b));
   }
   void add(int a,int b,int c)    {
System.out.println("sum of 3="+ (a+b+c));
   }
    }
    class OverLoad
    {   public static void main(String[] args)
   { Sample s=new Sample ( );
    s.add (20, 25);
    s.add (20, 25, 30);
   }     }
```

Output: sum of two=45
Sun of three=75

o Method Overriding: Writing two or more methods in super & sub classes **with same name and same signatures is called method overriding.**

WAP that contains a super and sub class which contains a method with same name and same method signature, behavior of the method is dynamically decided.

// **Dynamic polymorphism**: **overriding of methods**

```
    class Parent
    {
       void move()
      {
System.out.println ("Parent can move");
       }
    }
    class Child extends Parent
    {
       void move()
        {
System.out.println ("Child can walk & run");
        }
    }
  class OverRide
    {
      public static void main(String args[])
       {
Parent a = new Parent ( );
Parent b = new Child (); // Animal reference
                but Dog object (Up casting)
  a.move (); // runs method in Animal class
  b.move (); //Runs method in Dog class
 }
    }
```

Output: Parent can move
Child can walk & run

Static Polymorphism: The polymorphism exhibited at compile time is called Static polymorphism. Here the compiler knows which method is called at the compilation. This is also called **compile time polymorphism or static binding**.

Achieving method overloading & method overriding using private, static and final methods is an example of Static Polymorphism.

Write a program to illustrate static polymorphism.

```
//Static Polymorphism
  class Parent
   {   static void move ()
{ System.out.println ("Parent method");
}
    }
    class Child extends Parent
    {      static void move ()
{ System.out.println ("Child method");
}
    }
    public class StaticPoly
    {   public static void main(String args[])
{
     Parent.move ( ); //Calling static method with class name
     Child.move ( );     //Calling static method with class name
}
    }
```

Output: Parent method
Child method

Runtime polymorphism or Dynamic Method Dispatch

Runtime polymorphism or **Dynamic method dispatch** is the mechanism by which a call to **an overridden method is resolved at runtime, rather than compile time**. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Method to execution based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at runtime. In this process, an overridden method is called **through the reference variable of a super class**. The determination of the method to be called is based on the object being referred to by **the reference variable**.

Let's first understand the **upcasting** before Runtime Polymorphism.

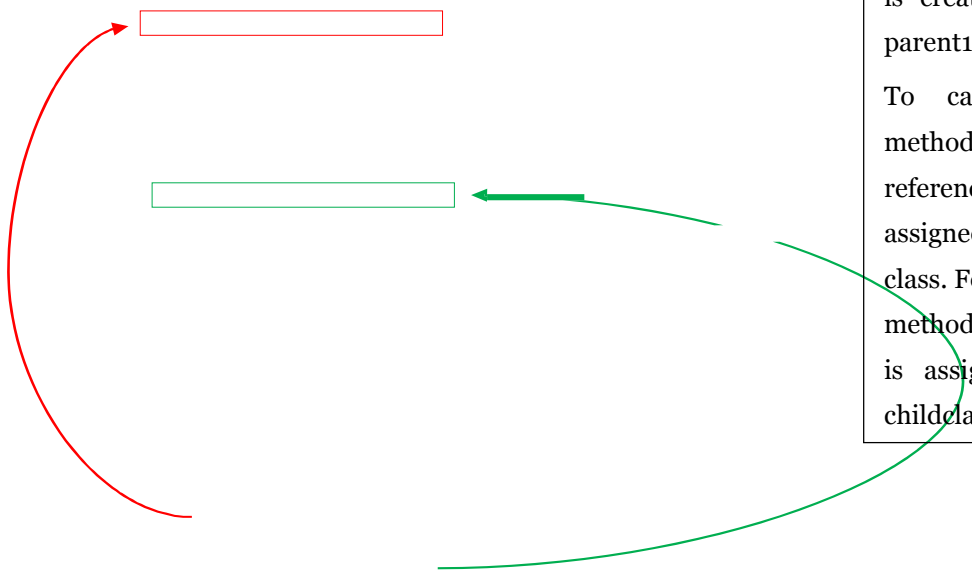
Upcasting

When reference variable of Parent class refers to the object of Child class, it is known as upcasting.

For example:

```
class A{}
class B extends A{}
A a=new B();//upcasting
```

Example



In this example a reference is created "***Ref1***" of type parent1.

To call an overridden method of any class this reference variable is assigned an object of that class. For ex. To call **sum()** method of child class Ref1 is assigned object **c1** of childclass.

Output

Real time example of Java Runtime Polymorphism

Consider a scenario; Bank is a class that provides method to get the rate of interest. But, rate of interest may differ according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.

```
EX1: class Bank{
  int getRateOfInterest(){return 0;}
  }
  class SBI extends Bank{
  int getRateOfInterest(){return 8;}
  }
  class ICICI extends Bank{
  int getRateOfInterest(){return 7;}
  }
  class AXIS extends Bank{
  int getRateOfInterest(){return 9;}
  }
  class Test3{
  public static void main(String args[]){
  Bank b1=new SBI();        Bank b2=new ICICI();        Bank b3=new AXIS();
  System.out.println("SBI Rate of Interest: "+b1.getRateOfInterest());
  System.out.println("ICICI Rate of Interest: "+b2.getRateOfInterest());
  System.out.println("AXIS Rate of Interest: "+b3.getRateOfInterest());
  } }
```

Output: SBI Rate of Interest: 8 ICICI Rate of Interest: 7 AXIS Rate of Interest: 9

Rule: Runtime polymorphism can't be achieved by data members.

```
Ex2:
class Bike{
 int speedlimit=90;
}
class Honda3 extends Bike{
 int speedlimit=150;

public static void main(String args[]){
    Bike obj=new Honda3();
System.out.println(obj.speedlimit);//90
}
```

Output:90

Difference between method overloading and method overriding in java

There are many differences between method overloading and method overriding in java. A list of differences between method overloading and method overriding are given below:

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

6)	***Java Method Overloading example***
	class OverloadingExample{
	static int add(**int** a,**int** b){
	return a+b;
	}
	static int add(**int** a,**int** b,**int** c){
	return a+b+c;
	} }
		Java Method Overriding example
		class Animal{
		void eat(){System.out.println("eating...");
		}
		}
		class Dog **extends** Animal{
		void eat(){
		System.out.println("eating bread...");} }

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user. Another way, it shows only important things to the user and hides the internal details **for example** sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Ways to achieve Abstaction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)
3.

Note1: It can be applied to methods and classes. It cannot be used with data members.
Note2: It needs to be extended and its method implemented. It cannot be instantiated.

Abstract class in Java

A class that is declared with **abstract keyword**, is known as abstract class in java. It can have **abstract** and **non-abstract methods** (method with body).

Example abstract class

abstract class A{}

1. Abstract class is a class that cannot be instantiated. (no creation of object).
2. It can be / must be inherited.
3. It can act as only base class in inheritance hierarchy. (levels)

Abstract method

A method that is declared as **abstract and does not have implementation** is known as **abstract method.**

1. It is a method, which has no body (definition) in the base class. The body should be implemented in the sub class only.
2. Any class with at least one abstract method should be declared as abstract.
3. If the sub class is not implementing an abstract method, its sub class has to implement it.

Example abstract method

abstract void printStatus(); //no body and abstract

Example of abstract class that has abstract method

In this example, Bike the abstract class that contains only one abstract method run. It implementation is provided by the Honda class.

```
abstract class Parent{
abstract void run();
 }
class Child extends Parent{
   void run(){System.out.println("Child
calling..");}
 public static void main(String args[]){
Parent obj = new Child();
obj.run();
}
}
```

Output: Child calling..

Note: With respect to a class, abstract and final are exclusive. They both cannot be applied to a class of a time.

```
abstract final class A
{
}
```

(The above code is wrong)

File: TestAbstraction1.java

```
abstract class Shape{
abstract void draw();
}
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");
}
}
class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();//In real scenario, object is pro
vided through method e.g. getShape() method
s.draw();
}
}
```

Output: drawing circle

File: TestBank.java

```
abstract class Bank{
abstract int getRateOfInterest();
}
class SBI extends Bank{
int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
int getRateOfInterest(){return 5;}
}
class TestBank{
public static void main(String
                          args[ ]){
Bank b=new SBI();
int interest=b.getRateOfInterest();
System.out.println("Rate of Interest
is: "+interest+" %");
}}
```

Output: Rate of Interest is: 7 %

Abstract class having constructor, data member, methods etc.
An abstract class can have data member, abstract method, method body, constructor and even main() method.

File: TestAbstraction2.java

```
//example of abstract class that have method body
abstract class Bike{
  Bike( ){

System.out.println("bike is created");}
        abstract void run( );
       void changeGear( ){
      System.out.println("gear changed");}
}
 class Honda extends Bike{
void run( ){

System.out.println("running safely..");}
 }
 class TestAbstraction2{
 public static void main(String args[ ]){
  Bike obj = new Honda();
  obj.run();
  obj.changeGear();
 }
}
```

Output: bike is created
running safely..
gear changed

Rule: *If there is any abstract method in a class, that class must be abstract.*

```
class Bike12{
abstract void run();
}
```

Output: compile time error

Rule: *If you are extending any abstract class that has abstract method, you must either provide the implementation of the method or make this class abstract.*

Interface in Java

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.

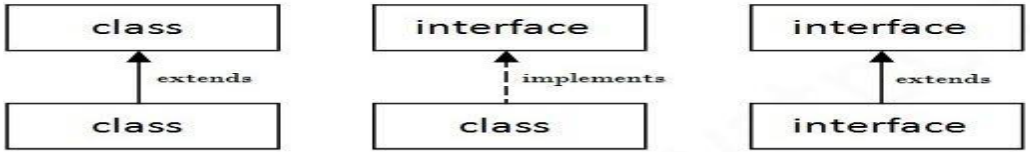
Java Interface also **represents IS-A relationship**.
It cannot be instantiated just like abstract class.

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.

NOTE: The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

Understanding relationship between classes and interfaces



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

Simple example of Java interface

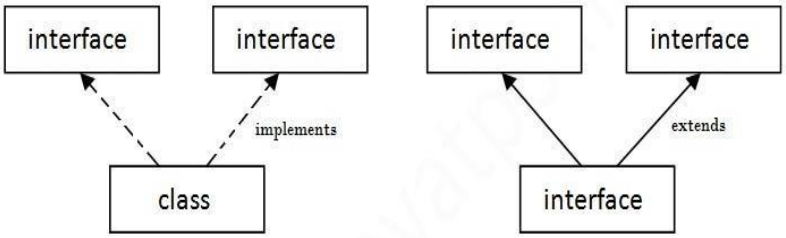
In this example, Printable interface have only one method, its implementation is provided in the A class.

```
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.




```
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

Interface inheritance

A class implements interface but one interface extends another interface .

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class Testinterface2 implements Showable{
  public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
public static void main(String args[]){
Testinterface2 obj = new Testinterface2();
obj.print();
obj.show();
 } }
```

Output: Hello
Welcome

A programmer uses an abstract class when there are some common features shared by all the objects. Interfaces are written when the programmer wants to leave the implementation to third party vendors. An interface is a specification of method prototypes. All the methods in an interface are abstract methods.

- An interface is a specification of method prototypes.
- An interface contains zero or more abstract methods.
- All the methods of interface are public, abstract by default.
- Once an interface is written any third party vendor can implement it.
- All the methods of the interface should be implemented in its implementation classes.
- If any one of the method is not implemented, then that implementation class should be declared as abstract.
- **We cannot create an object to an interface.**
- We can create a reference variable to an interface.
- **An interface cannot implement another interface.**
- An interface can extend another interface.
- **A class can implement multiple interfaces.**

Defining an interface:

An interface is defined much like a class.

This is the general form of an interface:

```
access interface name
    {
return-type     method-name1(parameter-list);
return-type     method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
// ...
return-type method-nameN(parameter-list);
type final-varnameN = value;
}
```

Here is an example of an interface definition. It declares a simple interface that contains one method called callback() that takes a single integer parameter.

```
interface Callback
{
void callback(int param);
}
```

Implementing Interfaces

Once an interface has been defined, one or more classes can implement that interface. The general form of a class that includes the implements clause looks like this:

class classname [extends superclass] [implements interface [,interface...]]

```
{
// class-body
}
```

If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared public.

Accessing Implementations through Interface References:

You can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them. The calling code can dispatch through an interface without having to know anything about the "callee."

The following example calls the callback( ) method via an interface reference variable:

```
class TestIface
{
public static void main(String args[])
{
Callback c = new Client();
c.callback(42);
}
}
```

Output: callback called with 42

```
 // Another implementation of Callback.
class AnotherClient implements Callback
{
  // Implement Callback&#39;s interface
public void callback(int p)
{
System.out.println("Another version of callback");
System.out.println("p squared is "+ (p*p));
}
}
```

Now, try the following class:

```
class TestIface2
{
public static void main(String args[])
{
Callback c = new Client();
AnotherClient ob = new AnotherClient();
c.callback(42);
c = ob; // c now refers to AnotherClient object
c.callback(42);
}
}
```

Output: callback called with 42

```
Another version of callback p
squared is 1764
```

As you can see, the version of callback( ) that is called is determined by the type of object that c refers to at run time.

EXTENDING INTERFACES:

One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Following is an example: // One interface can extend one or more interfaces..

```
interface A {
void meth1();
void meth2();
}
                                        // B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
void meth3();
}
// This class must implement all of A and B
class MyClass implements B {
public void meth1() {
System.out.println("Implement meth1().");
}
public void meth2() {
System.out.println("Implement meth2().");
}
public void meth3() {
System.out.println("Implement meth3().");
}
}
class IFExtend {
public static void main(String arg[]) {
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
}
}
```

As an experiment, you might want to try removing the implementation for meth1( ) in MyClass. This will cause a compile-time error. As stated earlier, any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated. But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract**methods.	Interface can have **only abstract** methods.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can have static methods, main method and constructor**.	Interface **can't have static methods, main method or constructor**.

5) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
6) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

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

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

Example of abstract class and interface in Java

```
//Creating interface that has 4 methods
interface A
{
void a();//bydefault, public and abstract
void b();
void c();
void d();
}
 //Creating abstract class that provides the implementation of one method of A interface
abstract class B implements A
  {
    public void c(){System.out.println("I am C");
   }
}
//Creating subclass of abstract class, now we need to provide the implementation of methods
class M extends B
{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}
//Creating a test class that calls the methods of A interface
class Test5
{
  public static void main(String args[])
  {
    A a=new M();
    a.a();
    a.b();
    a.c();
    a.d();
   }
}
```

Output:

I am a I am b I am c I am d

Packages in JAVA:

Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. **You can define classes inside a package that are not accessible by code outside that package.** You can also define class members that are only exposed to other members of the same package.

A **java package** is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.
4) A group of package called a library. The classes and interfaces of a package are likes books in a library and can be reused several times.

Defining a Package:

• Creating a package is quite easy: simply include **a package command as the first statement in a Java source file.** Any classes declared within that file will belong to the specified package. The package statement defines a name space in which classes are stored.

Creating a Package:

•The general form of the package statement:

package pkg; *//Here, pkg is the name of the package.*

• For example, the following statement creates a package called MyPackage.

package MyPackage;

The **package keyword** is used to create a package in java.

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

Note: Remember that the directory name must match the package name exactly.

More than one file can include the same package statement. The package Statement simply specifies to which package the classes defined in a file belong. You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period.

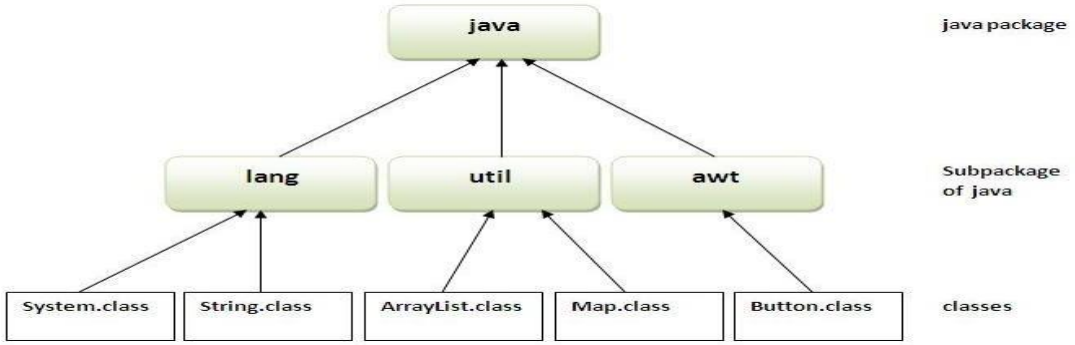
•The general form of a multileveled package statement is shown here:

package pkg1[.pkg2[.pkg3]];

•A package hierarchy must be reflected in the file system of your Java development.

For example, a package declared as package java.awt.image; needs to be stored in java:awt:image on your UNIX, Windows, or Macintosh file system, respectively.

•You cannot rename a package without renaming the directory in which the classes are stored.




```
//save as Simple.java
package mypack;
public class Simple{
 public static void main(String args[]){
   System.out.println("Welcome to package");
  }
}
```

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

javac -d directory javafilename

For **example:** javac -d . Simple.java

The **-d switch specifies the destination where to put the generated class file.** You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the **same directory, you can use . (dot).**

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Output: Welcome to package

Note: The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access package from another package?

There are three ways to access the package from outside the package.

1. **import package.*;**
2. **import package.classname;**
3. **fully qualified name.**

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages. The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

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

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

```
javac -d . A.java
javac -d . B.java
java  mypack.B
```

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible. Example of package by import package.classname

```
//save by A.java
package   pack;
public class A{
  public void msg(){System.out.println("
Hello1");
}
}
```

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

class B1{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Output: Hello1

```
javac -d . A.java
javac -d . B1.java
java  mypack.B1
```

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello2");    }
}
//save by B.java
package mypack;
class B2{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

Output: Hello

```
javac -d . A.java
javac -d . B2.java
java  mypack.B2
```

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

For example, consider the following package specification:
package MyPack;

In order for a program to find MyPack, one of two things must be true. Either the program is executed from a directory immediately above MyPack, or CLASSPATH must be set to include the path to MyPack.

The first alternative is the easiest (and doesn,,t require a change to CLASSPATH), but the second alternative lets your program find MyPack no matter what directory the program is in.

Create the package directories below your current development directory, put the .class files into the appropriate directories and then execute the programs.

Example:

```
// A simple package
package bal;
class Balance
    {
String name;
            double bal;
            Balance(String n, double b)
           {
name = n;
bal = b;
}
void show()
{
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}       }
class AccountBalance
    {
public static void main(String args[])
{
Balance current[] = new Balance[3]; current[0]
= new Balance("Kalpana", 123.23); current[1] =
new Balance("Sudha", 157.02); current[2] =
new Balance("Varshi", -12.33);
   for(int i=0; i<3; i++)
    current[i].show();
}
}
```

```
javac –d . AccountBalance.java
java bal.AccountBalance
OUTPUT:
Kalpana: $123.33
Sudha: $1577.02
-->Varshi: $-12,33
```

Save this file as , and put it in a directory called MyPack. Next, compile the file. Make sure that the resulting .class file is also in the MyPack directory. Executing the AccountBalance class, using the following command line:

java bal.AccountBalance

Remember, we should be in the directory above MyPack when you execute this command, or to have your CLASSPATH environmental variable set appropriately.

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

java AccountBalance *(AccountBalance must be qualified with its package name.)*

Importing Packages:

Java includes the import statement to bring certain classes, or entire packages, into visibility. The import statement is convenient. In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions.

This is the general form of the import statement:
import pkg1[.pkg2].(classname|*);

Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.). Finally, you specify either an explicit **classname or a star (*), which indicates that the Java compiler should import the entire package.**

Ex - This code fragment shows both forms in use:

```
import java.util.Date;
import java.io.*;
```

It is a good idea to explicitly name the classes that you want to use rather than importing whole packages. However, the **star form** has absolutely no effect on the run-time performance or size of your classes.

For example, /* Now, the Balance class, its constructor, and its show() method are public. This means that they can be used by non-subclass code outside their package.*/

Balance.java

```
package bal;
public class Balance
{
String name;
double bal;
public Balance(String n, double b)
{
name = n;
bal = b;
}
public void show()
{
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
```

For example, here TestBalance imports bal package and is then able to make use of the Balance class: **TestBalance.java**

```
import bal.Balance;
class TestBalance
{
public static void main(String args[])
{
/* Because Balance is public, you may use Balance
class and call its constructor. */
Balance test = new Balance("Kalpana",
99.88);
test.show( );
} }
```

```
Javac –d . Balance.java
Javac TestBalance.java
Java TestBalance
```

O/P; Kalpana: $99.88

Java API Packages:

Reuse of code is one of the important features of OOP. We can achieve this by extending the classes and implementing the interfaces which we have created. But this is limited to reusing the classes within a program. A package is a collection of classes and interfaces which provides a high level access protection and name space management.

Java API packages or Java* Built-in packages*:

Java API provides a huge number of classes grouped into various packages according to functionality. Now we will see important java API packages and the classes belonging to these packages.

1. **Java.lang:** This package contains language support classes. These are the classes which java compiler itself uses and therefore they are imported automatically. They include classes for primitive types, math functions, strings, StringBuffer, StringTokenizer, threads and exceptions.
2. **Java.uitl:** This package contains language utility classes like hash tables, vectors, date, Stack, Random etc.,
3. **Java.applet:** This package contains classes for creating and implementing applets.
4. **Java.net:** This package contains classes for networking. They not only include classes for communicating with local computers but also with internet services.
5. **Java.awt:** This package contains classes for implementing GUI. They include classes for Frame, Graphics,Buttons, windows, lists etc.

6. **Java.io:** This package contains I/O support classes. They provide facilities for input and output of data. They include classes for BufferedInputStream, BufferedOutputStream, DataInputStream and DataOutputStream.
7. **Javax.swing**: this package helps to develop GUI like java.awt. The „x" in javax represents that it is an extended package which means it is a package developed from another package by adding new features to it. In fact, javax.swing is an extended package of java.awt.

User-Defined packages:

Just like the built in packages shown earlier, the users of the Java language can also create their own packages. They are called user-defined packages. User-defined packages can also be imported into other classes and used exactly in the same way as the Built-in packages.

Packages Examples:

Program 1: Write a program to create a package pack with Addition class.

```
//creating a package
package pack;
public class Addition
{ private double d1,d2;
    public Addition(double a,double b)
    { d1 = a;
      d2 = b;
    }
    public void sum()
    { System.out.println ("Sum of two given numbers is : " + (d1+d2) );
    }
}
```

Compiling the above program:

Program 2: Write a program to use the Addition class of package pack.

```
//Using the package pack
import pack.Addition; class
Use
{ public static void main(String args[])
    { Addition ob1 = new Addition(10,20);
      ob1.sum();
    }
}
```

Output:

Program 3: Write a program to add one more class Subtraction to the same package pack.

```
//Adding one more class to package pack:
package pack;
public class Subtraction
{
```

```
    private double d1,d2;
    public Subtraction(double a, double b)
    {
      d1 = a;
      d2 = b;
    }
    public void difference()
    { System.out.println ("Sum of two given numbers is : " + (d1 - d2) );
    }
}
```

Compiling the above program:

Program 4: Write a program to access all the classes in the package pack.

```
//To import all the classes and interfaces in a class using import pack.*;
import  pack.*;
class Use
{ public static void main(String args[])
    { Addition ob1 = new Addition(10.5,20.6);
      ob1.sum();
      Subtraction ob2 = new Subtraction(30.2,40.11);
      ob2.difference();
    }
}
```

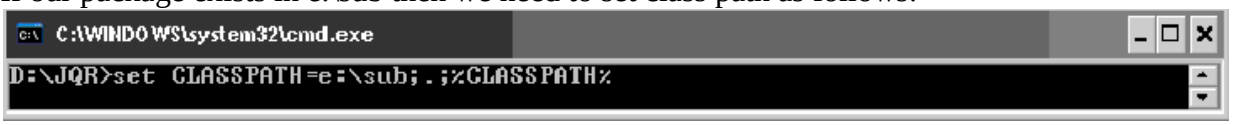
Output:

UNDERSTANDING CLASSPATH:

If the package pack is available in different directory, in that case the compiler should be given information regarding the package location by mentioning the directory name of the package in the classpath.

The CLASSPATH is an environment variable that tells the Java compiler where to look for class files to import.

If our package exists in e:\sub then we need to set class path as follows:



We are setting the classpath to e:\sub directory and current directory (.) an %CLASSPATH% means retain the already available classpath as it is.

Creating Sub package in a package: We can create sub package in a package in the format:

```
package packagename.subpackagename;
```

.: package pack1.pack2;

Here, we are creating pack2 subpackage which is created inside pack1 package. To use the classes and interfaces of pack2, we can write import statement as:

import pack1.pack2; **Ex: import java.io.*;**

Program 5: Program to show how to create a subpackage in a package.

```
//Creating a subpackage in a package
package pack1.pack2;
public class Sample
{
     public void show ()
    {
      System.out.println ("Hello Java Learners");
    }
}
```

ACCESSING A PACKAGES:

Access Specifier: Specifies the scope of the data members, class and methods.

- private members of the class are available with in the class only. The scope of private members of the class is "CLASS SCOPE".
- public members of the class are available anywhere . The scope of public members of the class is "GLOBAL SCOPE".
- default members of the class are available with in the class, outside the class and in its sub class of same package. It is not available outside the package. So the scope of default members of the class is "PACKAGE SCOPE".
- protected members of the class are available with in the class, outside the class and in its sub class of same package and also available to subclasses in different package also.

Class Member Access	private	No Modifier	protected	public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Another Simple example for Packages:

Op.java

```
package my;
public claa Op
{
public void add(int a,int b)
  {
int c=a+b;
System.out.println("Add:"+C);
  }
}
```

Demo.java

```
import my.Op;
class Demo
{
public static void
          main(String args[])
{
  Op obj=new Op();
    obj.add(10,20);
    }
}
```

Compilation

javac –d . Op.java
javac Demo.java

Run

java Demo

Op:

Add:30

DIFFERENCES BETWEEN CLASSES AND INTERFACES:

Classes	Interfaces
Classes have instances as variables and methods with body	Interfaces have instances as abstract methods and final constants variables.
Inheritance goes with extends keyword	Inheritance goes with implements keywords.
The variables can have any acess specifier.	The Variables should be public, static, final
Multiple inheritance is not possible	It is possible
Classes are created by putting the keyword class prior to classname.	Interfaces are created by putting the keyword interface prior to interfacename(super class).
Classes contain any type of methods. Classes may or may not provide the abstractions.	Interfaces contain mustly abstract methods. Interfaces are exhibit the fully abstractions

Differences between object and class

SNo.	Object	Class
1)	Object is an **instance** of a class.	Class is a **blueprint or template** from which objects are created.
2)	Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc.	Class is a **group of similar objects**.
3)	Object is a **physical** entity.	Class is a **logical** entity.
4)	Object is created through **new keyword** mainly e.g. Student s1=new Student();	Class is declared using **class keyword** e.g. class Student{}
5)	Object is created **many times** as per requirement.	Class is declared **once**.
6)	Object **allocates memory when it is created**.	Class **doesn't allocated memory when it is created**.
7)	There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization.	There is only **one way to define class** in java using class keyword.

UNIT III: Exception handling - Concepts of exception handling, benefits of exception handling, exception hierarchy, usage of try, catch, throw, throws and finally, built in exceptions, creating own exception sub classes.

Multi threading: Differences between multi threading and multitasking, thread life cycle, creating threads, thread priorities, synchronizing threads, inter thread communication, daemon threads.

Exceptions:

•An exception (or exceptional event) is **a problem** or **an abnormal condition** that arises during the execution of a program or **at run time**. In other words an exception is a run time error.

When an Exception occurs the normal flow of the program is disrupted and the program terminates abnormally, therefore these exceptions are to be handled. An exception can occur for many different reasons, including the following:

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

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

A java exception is an object that describes an exceptional condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. Now the exception is caught and processed.

Errors and Exception Handling

No matter how good a program we write it may cause the following types of errors.

1. Syntax Errors
2. Logical Errors
3. Runtime Errors

Syntax Errors: It is an error which occurs when the rules of language are violated . If we are not following the rules, those cause the error. It can be easily detected and corrected by compilers. **Eg:** Statement missing **semicolon;** , Missing/expected **„ }"** ,Un-defined symbol **varname** etc.

Logical Error: It is an error caused due to manual mistake in the logical expressions. These errors cannot be detected by the compiler.
Eg. Netsal = (basic+da+hra) (+) pf

Runtime Error: It is an Error, which occurs after executing the program at runtime.
Eg. Divide by zero.
File not found
No disc in the drive etc.

Exception:

Exceptions are an unusual condition that occurs at runtime and can lead to errors that can crash a program. It is nothing but **runtime error**. It can be handled to provide a suitable message to

user.

Exception Handler:

It is a piece of code; **Exceptions** are conditions within the code. A developer can handle such conditions and take necessary corrective actions. Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, 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:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
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 statement will be executed.

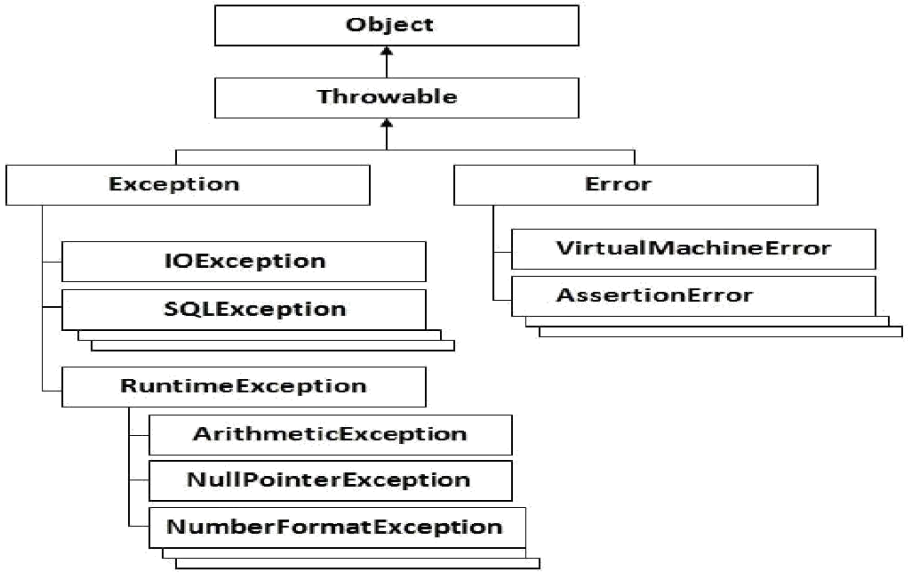
It throws an exception whenever a calling method encounters an error providing that the calling method takes care of that error.

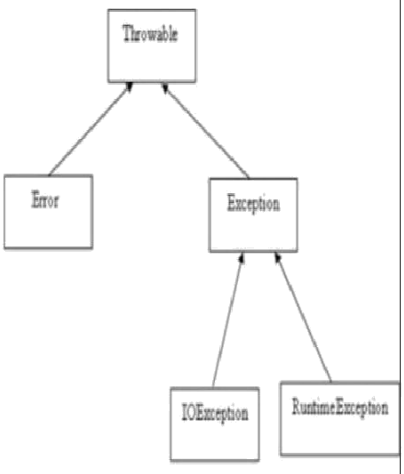
It also gives us the scope of organizing and differentiating between different error types using a separate block of codes. This is done with the help of try-catch blocks.

Hierarchy of Java Exception classes

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

The Exception class has two main subclasses: IOException class and RuntimeException Class.





IOException:
It is an Exception class. It is a part of java.io packages. It is required to be caught in every method when readLine() is used. Else throws class is used to mention IOException.

Runtime Exceptions: A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.

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

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.

Differences between Checked and Unchecked Exceptions

Checked Exceptions	Unchecked Exceptions
Checked Exceptions are the sub-class of the Exception class	Unchecked Exceptions are the sub-class of the Runtime Exceptions
In Checked Exceptions the Java Virtual Machine requires the exception to be caught or handled	In Unchecked Exceptions the Java Virtual Machine does not require the exception to be caught or handled
Checked Exceptions are checked at compile time of the program	Unchecked Exceptions are checked at the run time or during execution time of the program
Checked Exceptions can be created manually	Unchecked Exceptions can be created manually.
Checked Exceptions can be handled using try, catch and finally	Unchecked Exceptions can be handled using try, catch and finally
Checked Exceptions are mainly illegal path or illegal use.	Unchecked Exceptions are mainly programming mistakes or bad input
Checked Exceptions can be ignored in a program	Unchecked Exceptions cannot be ignored in a program.

Checked exceptions must be mentioned in the throws clause of all methods in which they could occur.	Unchecked exceptions usually not specified in the throws clause of a method and it mentioned in the try–catch-finally.
It uses Mainly when an exception occurs and we know what we need to do and Where chances of failure are greater.	It uses Business logic, Rules validation and User input validation.

import java.io.*;
class Example {
public static void main(String
args[]) throws IOException
{
FileInputStream fis = null;
fis = new
FileInputStream("B:/myfile.txt");
int k;
while((k = fis.read()) != -1) {
System.out.print((char)k); }
fis.close();
} }
Output:
File content is displayed on the screen.
	class Example {
	public static void main(String args[])
	{
	try{
	int arr[] ={1,2,3,4,5};
	System.out.println(arr[7]);
	}
	catch(ArrayIndexOutOfBoundsException e)
	{
	System.out.println("The specified index does
	not exist " +"in array. Please correct the
	error.");
	}
	}
	}

Exception class:

This is the generic catch because exception class can handle any kind of exception. Hence, it should be kept at the end of all catches to ensure that no exception will go un-defined on running the prg.

Built-in Exception classes:

Java defines several exception classes inside the standard package **java.lang**. The most general of these exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available.

Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java Unchecked RuntimeException.

Exception	Description
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.

Exception	Description
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.

Following is the list of Java Checked Exceptions Defined in java.lang.

Exception	Description
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

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
```

Exception Handling Mechanism in Java:

Java uses a mechanism or model to handle exceptions. This model is known as **"catch and throw model"**. This means that "it identifies an exception and throws it and it will be caught by exception handler". This model used the following keywords.

Try, catch, throw, throws and finally

Try block:

All statements that are likely to raise an exception at run time are kept in try block. This block will detect an exception and throws it. It will be handled by catch block.

Catch:

This block takes corrective steps when an exception is thrown. It provides a suitable message to the user. Then user will be able to proceed with the application.

A try block can follow multiple catch blocks.

Throw:

It is used to throw an exception.

Throws:

It is used to specify the possible exceptions that are caused by a method. But not handled by the method.

Finally:

This block is followed by catch block. This used to close files, data base connections etc. This block is executed irrespective of whether exception occurs or not.

```
try{
  //Do something
}
catch(Exception ex)
{
  //Catch exception hare using exception argument ex
}
```

The code which is prone to exceptions is placed in the try block, when an exception occurs; that exception occurred is handled by catch block associated with it.

Example:

```
class ExceptionSample {
    public static void main (String[] arrgs) // Main function
  {
    int no1, no2;         //Data variables
    no1=5;           //Contain some value
    no2=0;           //contain some value
     try                 // Try block
    {
      System.out.println(no1/no2);    //attempt to divide number by 0
    }
    catch (Exception ex)    // Catch block receive exception
    {/* Display exception message */ System.out.println("Error
      with defination: "+ex.getMessage());
     }        }   }
```

User-defined exception (or) Custom Exception (or) Own Exception:

The built-in exceptions do not meet our specific requirements. We can define our own exception classes such exceptions are known, as user-defined exception must extend the base class exception.

Syn.: class classname extends Exception

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

Example:

```
class InvalidAgeException extends Exception
    {
 InvalidAgeException(String s)
       {
         super(s);
        }
    }

class TestCustomException1
{
static void validate(int age)throws InvalidAg
eException
  {
    if(age<18)
throw new InvalidAgeException("not valid");

  else
     System.out.println("welcome to vote");
       }
   public static void main(String args[ ])
{
     try
    {
     validate(13);
     }
    catch(Exception m)
     {
 System.out.println("Exception occur: "+m);
      }
 System.out.println("rest of the code...");
  }
}
```

Output:

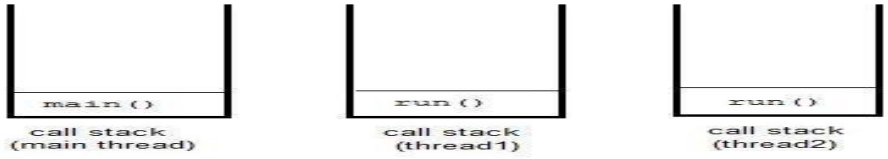
```
Exception occured:
InvalidAgeException:not valid
            rest of the code...
```

By the help of custom exception, you can have your own exception and message. Let's see a simple example of java custom exception.

Multithreading

A program can be divided into a number of small processes. Each small process can be addressed as a single thread (a lightweight process). Multithreaded programs contain two or more threads that can run concurrently. This means that a single program can perform two or more tasks simultaneously. For example, one thread is writing content on a file at the same time another thread is performing spelling check.

An instance of **Thread** class is just an object, like any other object in java. But a thread of execution means an individual "lightweight" process that has its own call stack. In java each thread has its own call stack.



A Process consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist on its own; It must be a part of process.

There are two distinct types of Multitasking i.e., Processor-based and Thread-based Multitasking.

What is the difference between thread based and processor-based Multitasking

Thread	Process
Thread is a light-weight process.	Process is a heavily-weight process.
Threads share the address space	Processes require their own separate address space.
In Thread-based multitasking, thread is the smallest unit of code, which means a single program can perform two or more tasks simultaneously. For example a text editor can print and at the same time you can edit text provided that those two tasks are perform by separate threads.	Process-Based multitasking is a feature that allows your computer to run two or more programs concurrently. For example you can listen to music and at the same time chat with your friends on Facebook using browser.
Inter thread communication is inexpensive, and context switching from one thread to the next is lower in cost.	Inter process communication is expensive and limited

Difference between Multi tasking and Multi threading

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

There are two distinct types of multitasking: **process-based and thread-based**.

A **process** is a program that is executing. Thus, process-based multitasking is the feature that allows to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor.

In **process-based multitasking,** a program is the smallest unit of code that can be dispatched by the scheduler.

In a **thread-based multitasking** environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

Thus, process-based multitasking deals with the "big picture," and thread-based multitasking handles the details.

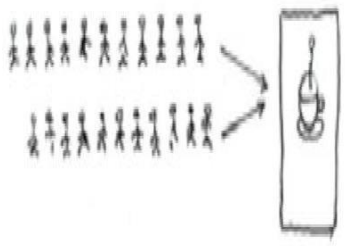
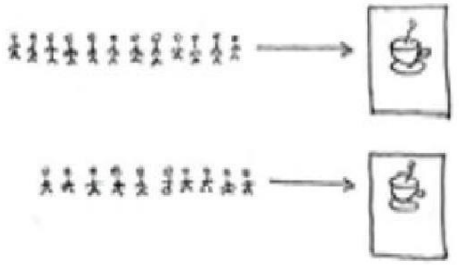
Multitasking threads require less overhead than multitasking processes. Processes are **heavyweight** tasks that require their own separate address spaces. Inter process communication is expensive and limited. Context switching from one process to another is also costly.

Threads, on the other hand, are **lightweight**. They share the same address space and cooperatively share the same heavyweight process. Inter thread communication is inexpensive, and context switching from one thread to the next is low cost.

Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. This is especially important for the interactive, networked environment in which Java operates, because idle time is common.

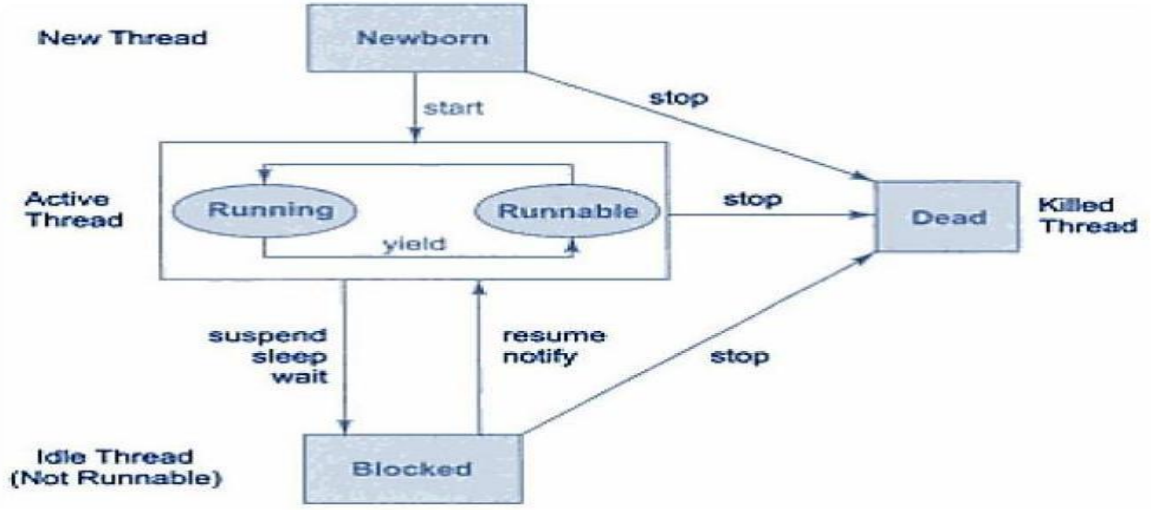
Benefits of Multithreading

1. Enables programmers to do multiple things at one time
2. Programmers can divide a long program into threads and execute them in parallel which eventually increases the speed of the program execution
3. Improved performance and concurrency
4. Simultaneous access to multiple applications

Concurrent	**Parallel**
– Tasks that overlap in time • The system might run them in parallel on multiple processors	– Tasks that run at the same time on different processors
Two Queues One coffee machine 	Two Queues Two coffee machines 

Definition: Thread is a tiny program running continuously. It is sometimes called as light-weight process.

Life cycle of a Thread



A thread can be in any of the five following states

1. Newborn State: When a thread object is created a new thread is born and said to be in Newborn state.
2. Runnable State: If a thread is in this state it means that the thread is ready for execution and waiting for the availability of the processor. If all threads in queue are of same priority then they are given time slots for execution in round robin fashion
3. Running State: It means that the processor has given its time to the thread for execution. A thread keeps running until the following conditions occurs

a. Thread give up its control on its own and it can happen in the following situations

i. A thread gets suspended using suspend() method which can only be revived with resume() method

ii. A thread is made to sleep for a specified period of time using sleep(time) method, where time in milliseconds

iii. A thread is made to wait for some event to occur using wait () method.

In this case a thread can be scheduled to run again using notify () method.

b. A thread is pre-empted by a higher priority thread

4. Blocked State: If a thread is prevented from entering into runnable state and subsequently running state, then a thread is said to be in Blocked state.
5. Dead State: A runnable thread enters the Dead or terminated state when it completes its task or otherwise terminates.

The main thread

Even if you don't create any thread in your program, a thread called **main** thread is still created. Although the**main** thread is automatically created, you can control it by obtaining a reference to it by calling**currentThread()** method.

Two important things to know about **main** thread are,

- It is the thread from which other threads will be produced.
- **main** thread must be always the last thread to finish execution.

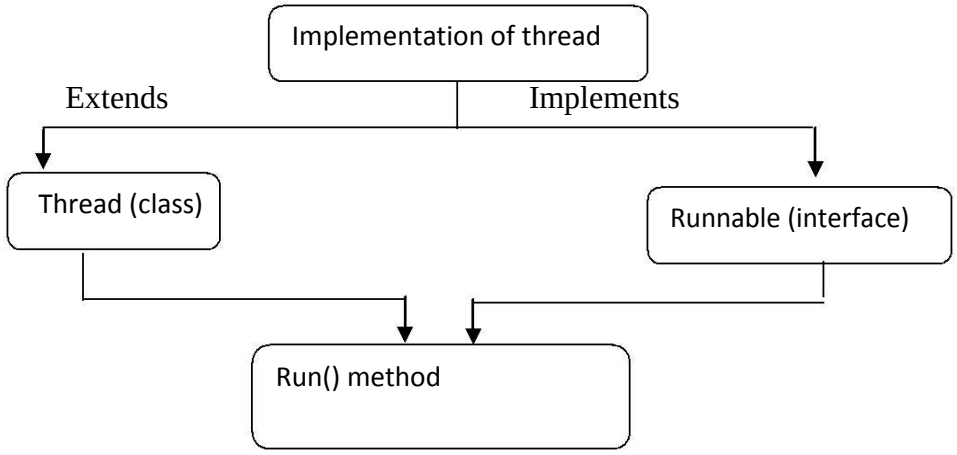
```
class MainThread
{
 public static void main(String[] args)
  {
  Thread t=Thread.currentThread();
  t.setName("MainThread");
  System.out.println("Name of thread is "+t);
   }
 }
```

Output : Name of thread is Thread[MainThread,5,main]

Creating a thread

Java defines two ways by which a thread can be created.

- By implementing the **Runnable** interface.
- By extending the **Thread** class.



Implementing the Runnable Interface

The easiest way to create a thread is to create a class that implements the runnable interface. After implementing runnable interface , the class needs to implement the **run()** method, which is of form,

```
public void run()
```

- run() method introduces a concurrent thread into your program. This thread will end when run() returns.
- You must specify the code for your thread inside run() method.
- run() method can call other methods, can use other classes and declare variables just like any other normal method.

```
EX: class T1 implements Runnable
    {
      public void run()
       {
          System.out.println("concurrent thread started running..");
        }
    public static void main( String args[] )
     {
       T1 obj = new T1();
       Thread t = new Thread(obj);
       t.start();
     }
   }
```

Output :

```
concurrent thread started running..
```

To call the **run()** method, **start()** method is used. On calling start(), a new stack is provided to the thread and run() method is called to introduce the new thread into the program.

Extending Thread class

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of the new thread.

```
class T2 extends Thread
{
 public void run()
 {
  System.out.println("one thread created..");
 }
public static void main( String args[] )
 {
  T2 obj1 = new T2();
  Obj1.start();
 }
}
```

Output :

```
one thread created..
```

In this case also, as we must override the **run()** and then use the **start()** method to start and run the thread.

Q: Can we start a thread twice?

Ans: No, if a thread is started it can never be started again, if you do so, an illegalThreadStateException is thrown. Example is shown below in which a same thread is coded to start again

Example:

```
class t2 extends Thread
{
    public void run()
    {
        System.out.println("Thread is Running");
    }
    public static void main(String args[])
    {
        t2 obj1 = new t2();
        obj1.start();
        obj1.start();
    }
}
```

As you can see two statements to start a same thread is written in the code which will not give error during compilation but when you run it you can see an Exception as shown in the Output Screenshot.

Output:

```
C:\NIEC Java>javac t2.java

C:\NIEC Java>java t2
Thread is Running
Exception in thread "main" java.lang.IllegalThreadStateException
        at java.lang.Thread.start(Unknown Source)
        at t2.main(t2.java:11)
```

Interrupting threads:

Stopping a thread: A thread can be stopped from running further by issuing the following statement-

th.stop();

By this statement the thread enters in a dead state. From stopping state a thread can never return to a runnable state.

Use of stop() Method

The stop() method kills the thread on execution

Example:

```
class C extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            if(i==2) stop();
            System.out.println("A:" +i);
        }
        System.out.println("Exit from A");
    }
    public static void main(String args[])
    {
        C c = new C();
        c.start();
    }
}
```

Condition is checked and when i==2 stop() method is evoked causing termination of thread execution

Output

```
C:\NIEC Java>java C
A:1
```

Blocking a thread:

A thread can be temporarily stopped from running. This is called blocking or suspending of a thread. Following are the ways by which thread can be blocked-

1. sleep( )

By **sleep** method a thread can be blocked for some specific time. When the specified time gets **elapsed** then the thread can return to a runnable state.

Example:

```
class C extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            try
            {
                if(i==2) sleep(1000);
            }
            catch(Exception e)
            {
            }
            System.out.println("A:" +i);
        }
        System.out.println("Exit from A");
    }
    public static void main(String args[])
    {
        C c = new C();
        c.start();
    }
}
```

Condition is checked and when i==2 sleep() method is evoked which halts the execution of the thread for 1000 milliseconds. When you see output there is no change but there is delay in execution.

Output

```
C:\NIEC Java>javac C.java

C:\NIEC Java>java C
A:1
A:2
A:3
A:4
A:5
Exit from A
```

2. suspend( )

By **suspend** method the thread can be blocked until further request comes. When the **resume()** method is invoked then the returns to a runnable stste.

```
class C extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            System.out.println("C:" +i);
        }
        System.out.println("Exit from C");
    }
}
class A extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            System.out.println("A:" +i);
        }
        System.out.println("Exit from A");
    }
}
class suspendtest
{
    public static void main(String args[])
    {
        C c = new C();
        A a = new A();
        c.start();
        a.start();
        c.suspend();
        c.resume();
    }
}
```

Although Thread 'C' is started earlier than Thread 'A' but due to suspend method Thread 'A' gets completed ahead of Thread 'C'

Output

```
C:\Achin Jain>java suspendtest
A:1
A:2
A:3
A:4
A:5
Exit from A
C:1
C:2
C:3
C:4
C:5
Exit from C
```

3. wait()

The thread can be made suspended for some specific conditions. When the **notify()** method is called then tha blocked thread returns to the runnable state.

The difference between the suspending and stopping thread is that if a thread is suspended then its execution is stopped temporarily and it can return to a runnable state. But in case, if a thread is stopped then it goes to a dead state and can never return to runnable state.

Thread Priorities

There are various properties of threads. Those are,

\} Thread priorities
\} Daemon Threads
\} Thread group

Thread Priorities:

Every thread has a priority that helps the operating system determine the order in which threads are scheduled for execution. In java thread priority ranges between,

1 MIN-PRIORITY (a constant of 1)
2 MAX-PRIORITY (a constant of 10)

By default every thread is given a NORM-PRIORITY(5).

The **main** thread always have NORM-PRIORITY.

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

Example:

```
public clicker(int p) {
t = new Thread(this);
t.setPriority(p);
}
public void run() {
while (running) {
click++;
}
}
public void stop() {
running = false;
}
public void start() {
t.start();
}
}
class HiLoPri {
public static void main(String args[]) {
Thread.currentThread().setPriority(Thread.
MAX_PRIORITY);
clicker hi = new
clicker(Thread.NORM_PRIORITY + 2);
clicker lo = new
clicker(Thread.NORM_PRIORITY - 2);
lo.start();
hi.start();
try {
Thread.sleep(10000);
} catch (InterruptedException e) {
System.out.println("Main thread
                    interrupted.");
}
lo.stop();
hi.stop();
   4  Wait for child threads to
      terminate. try {
hi.t.join();
lo.t.join();
} catch (InterruptedException e) {
System.out.println("InterruptedException
caught"); }System.out.println("Low-priority
thread: " + lo.click);
System.out.println("High-priority thread: " +
hi.click);

}
}
```

Output:

Low-priority thread: 4408112
High-priority thread: 589626904

EX2:

```
class prioritytest
{
    public static void main(String args[])
    {
        C c = new C();
        A a = new A();
        a.setPriority(10);
        c.setPriority(1);
        c.start();
        a.start();
    }
}
```

In the above code, you can see Priorities of Thread is set to maximum for Thread A which lets it to run to completion ahead of C which is set to minimum priority.

Output:

```
C:\Achin Jain>java prioritytest
A:1
A:2
A:3
A:4
A:5
Exit from A
C:1
C:2
C:3
C:4
C:5
Exit from C
```

Daemon Thread:

In Java, any thread can be a Daemon thread. Daemon threads are like a service providers for other threads. Daemon threads are used for background supporting tasks and are only needed while normal threads are executing.

If normal threads are not running and remaining threads are daemon threads then the interpreter exits. **setDaemon(true/false) ?** This method is used to specify that a thread is daemon thread.**public boolean isDaemon() ?** This method is used to determine the thread is daemon thread or not.

The core difference between user threads and daemon threads is that the JVM will only shut down a program when all user threads have terminated. Daemon threads are terminated by the JVM when there are no longer any user threads running, including the main thread of execution. Use daemons as the minions they are.

```
import java.io.*;
public class DaemonThread extends Thread
{
public void run()
{
System.out.println("Enter run method");
System.out.println("In run method: currentThread() is" +Thread.currentThread());
```

```
while(true)
{
  try
  {
   Thread.sleep(500);
   }
   catch(InterruptedException x)
  {
   }
System.out.println("In run method: woke up again");
  }
}
public static void main(String args[])
{
System.out.println("Entering main method");
DaemonThread t=new DaemonThread();
t.setDaemon(true);
t.start();
try
{
Thread.sleep(3000);
}
catch(InterruptedException x)
{
}
System.out.println("Leaving main thread");
} }
```

Output:

```
G:\Users\Varshitha Reddy\Desktop\java>javac DaemonThread.java

G:\Users\Varshitha Reddy\Desktop\java>java DaemonThread
Entering main method
Enter run method
In run method: currentThread() isThread[Thread-0,5,main]
In run method: woke up again
In run method: woke up again
In run method: woke up again
In run method: woke up again
In run method: woke up again
Leaving main thread
```

Synchronization

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

The synchronized keyword in Java creates a block of code referred to as a critical section. Every Java object with a critical section of code gets a lock associated with the object. To enter a critical section, a thread needs to obtain the corresponding object's lock.

```
synchronized(object)
{
  // statements to be synchronized
}
```

Problem without using Synchronization

In the following example method updatesum() is not synchronized and access by both the threads simultaneously which results in inconsistent output. Making a method synchronized, Java creates a "monitor" and hands it over to the thread that calls the method first time.

As long as the thread holds the monitor, no other thread can enter the synchronized section of the code. Writing the method as synchronized will make one thread enter the method and till execution is not complete no other thread can get access to the method.

```
class update
{
    void updatesum(int i)
    {
        Thread t = Thread.currentThread();
        for(int n=1; n<=5; n++)
        {
            System.out.println(t.getName()+" : "+(i+n));
        }
    }
}
class A extends Thread
{
    update u = new update();
    public void run()
    {
        u.updatesum(10);
    }
}
class syntest
{
    public static void main(String args[])
    {
        A a = new A();
        Thread t1 = new Thread(a);
        Thread t2 = new Thread(a);
        t1.setName("Thread A");
        t2.setName("Thread B");
        t1.start();
        t2.start();
    }
}
```

synchronized void updatesum(int i)

Output when method is declared as synchronized

```
C:\Achin Jain>java syntest
Thread A : 11
Thread A : 12
Thread A : 13
Thread A : 14
Thread A : 15
Thread B : 11
Thread B : 12
Thread B : 13
Thread B : 14
Thread B : 15
```

Output

```
C:\Achin Jain>java syntest
Thread A : 11
Thread B : 11
Thread A : 12
Thread B : 12
Thread A : 13
Thread B : 13
Thread A : 14
Thread B : 14
Thread A : 15
Thread B : 15
```

Note: The above output is using synchronized keyword.

Inter thread Communication

It is all about making synchronized threads communicate with each other. It is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter in the same critical section to be executed. These method are implemented as **final** in Object. All three method can be called only from within a **synchronized** context. It is implemented by the following methods of Object Class:

wait( ): This method tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ).

notify( ): This method wakes up the first thread that called wait( ) on the same object.

notifyAll( ): This method wakes up all the threads that called wait( ) on the same object. The highest priority thread will run first. These methods are implemented as final methods in Object, so all classes have them. All three methods can be called only from within a synchronized context.

Example:

```
class customer
{
int amount = 0;
int flag = 0;
public synchronized int withdraw(int amount)
{
    System.out.println(Thread.currentThread().getName()+" : is going to withdraw");
    if(flag==0)
    {
        try
        {
            System.out.println("Waiting....");
            wait();
        }
        catch(Exception e)
        {
        }
    }
    this.amount-=amount;
    System.out.println("Withdraw Complete");
    return amount;
}
public synchronized void deposit(int amount)
{
    System.out.println(Thread.currentThread().getName()+" : is going to Deposit");
    this.amount+=amount;
    notifyAll();
    System.out.println("Deposit Complete");
    flag=1;
}
}
```

If both these methods are commented which means there is no communication, output will be inconsistent. See **Output 2**

```
class threadcomm
{
    public static void main(String args[])
    {
        final customer c = new customer();
        Thread t1 = new Thread()
        {
            public void run()
            {
                c.withdraw(5000);
                System.out.println("After withdraw Amount is :"+ c.amount);
            }
        };
        Thread t2 = new Thread()
        {
            public void run()
            {
                c.deposit(10000);
                System.out.println("After Deposit Amount is :"+ c.amount);
            }
        };
        t1.start();
        t2.start();
    }
}
```

Output 1:

```
C:\Achin Jain>java threadcomm
Thread-0 : is going to withdraw
Waiting....
Thread-1 : is going to Deposit
Deposit Complete
After Deposit Amount is :10000
Withdraw Complete
After withdraw Amount is :5000
```

Output 2:

```
C:\Achin Jain>java threadcomm
Thread-0 : is going to withdraw
Waiting....
Withdraw Complete
After withdraw Amount is :-5000
Thread-1 : is going to Deposit
Deposit Complete
After Deposit Amount is :5000
```

Difference between wait() and sleep()

wait()	sleep()
called from synchronised block	no such requirement
monitor is released	monitor is not released
awake when notify() or notifyAll() method is called.	not awake when notify() or notifyAll() method is called
not a static method	static method
wait() is generaly used on condition	sleep() method is simply used to put your thread on sleep.

UNIT IV: Applets – Concepts of Applets, differences between applets and applications, life cycle of an applet, types of applets, creating applets, passing parameters to applets.

Event Handling- Events, Event sources, Event classes, Event Listeners, Delegation event model, handling mouse and keyboard events, Adapter classes.

Streams- Byte streams, Character streams, Text input/output.

APPLETS:

An applet is a program that comes from server into a client and gets executed at client side and displays the result.

An applet represents byte code embedded in a html page. (Applet = bytecode + html) and run with the help of Java enabled browsers such as Internet Explorer.

An applet is a Java program that runs in a browser. Unlike Java applications applets do not have a main () method.

To create applet we can use java.applet.Applet or javax.swing.JApplet class. All applets inherit the super class „Applet". An Applet class contains several methods that help to control the execution of an applet.

Advantages:

ιϖ. Applets provide dynamic nature for a webpage.
ϖ. Applets are used in developing games and animations.
ϖι. Writing and displaying (browser) graphics and animations is easier than applications.
ϖιι. In GUI development, constructor, size of frame, window closing code etc. are not required

Restrictions of Applets of Applets Vs Applications

6. Applets are required separate compilation before opening in a browser.
7. In realtime environment, the bytecode of applet is to be downloaded from the server to the client machine.
8. Applets are treated as **untrusted** (as they were developed by unknown people and placed on unknown servers whose trustworthiness is not guaranteed).
9. Extra Code is required to communicate between applets using **AppletContext**.

DEFFERENCES BETWEEN APPLETS AND APPLICATIONS

FEATURE	APPLICATION	APPLET
main() method	main() method Present	main() method Not present
Execution	Can be executed on standalone computer system. (JDK & JRE)	Used to run a program on client Browser like Chrome.

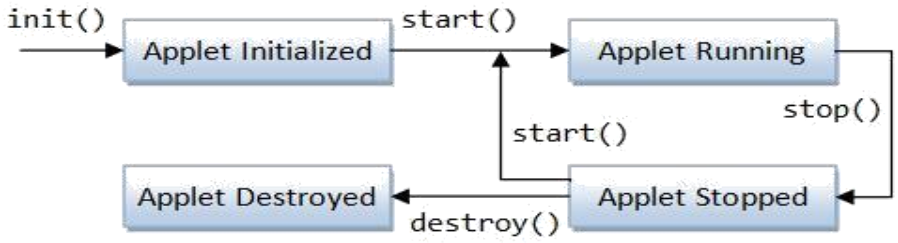
Nature	Called as stand-alone application as application can be executed from command prompt	Requires some third party tool help like a browser to execute

Restrictions	Can access any data or software available on the system	Cannot access anything on the system except browser‟s services
Security	Does not require any security	Requires highest security for the system as they are untrusted
Programming	larger programs	small programs
Platform	platform independent	platform independent
Accessibility	The java applications are designed to work with the client as well as server.	Applets are designed just for handling the client site problems.
Working	Applications are created by writing public static void main(String[] s) method	Applets are created by extending the java.applet.Applet class
Client side / Server side	The applications don't have such type of criteria	Applets are designed for the client site programming purpose
Methods	Application has a single start point which is main method	Applet application has 5 methods which will be automatically invoked.

Example	public class MyClass
	{
	public static void main(String args[]) {}
	}
		import java.awt.*;
		import java.applet.*;
		
		public
		class Myclass extends Applet
		{
		public void init()
		{
		}
		public void start()
		{
		}
		public void stop()
		{
		}
		public void destroy() {}
		
		public void paint(Graphics g) {}
		}

LIFE CYCLE OF AN APPLET

Let the Applet class extends Applet or JApplet class.



Initialization:

- **public void init():** This method is used for initializing variables, parameters to create components. This method is executed only once at the time of applet loaded into memory.

```
public void init()
{
//initialization
}
```

Runnning:

- **public void start ():** After init() method is executed, the start method is executed automatically. Start method is executed as long as applet gains focus. In this method code related to opening files and connecting to database and retrieving the data and processing the data is written.

Idle / Runnable:

- **public void stop ():** This method is executed when the applet loses focus. Code related to closing the files and database, stopping threads and performing clean up operations are written in this stop method.

Dead/Destroyed:

- **public void destroy ():** This method is executed only once when the applet is terminated from the memory.

Executing above methods in that sequence is called applet life cycle.

We can also use **public void paint (Graphics g)** in applets.

//An Applet skeleton.

```
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/
public class AppletSkel extends Applet {
 3 Called first.
    public void
    init() {
 4 initialization
}
```

```
/* Called second, after init(). Also called whenever the applet is restarted. */
public void start() {
3 start or resume execution
}
4 Called when the applet is
  stopped. public void stop() {
5 suspends execution
}
/* Called when applet is terminated. This is the last method executed. */
public void destroy() {
  5 perform shutdown activities
}
  6 Called when an applet's window must be
    restored. public void paint(Graphics g) {
  7 redisplay contents of window
}
}
```

After writing an applet, an applet is compiled in the same way as Java application but running of an applet is different.

There are two ways to run an applet.

- Executing an applet within a Java compatible web browser.
- Executing an applet using „appletviewer". This executes the applet in a window.

To execute an applet using web browser, we must write a small HTML file which contains the appropriate „APPLET" tag. <APPLET> tag is useful to embed an applet into an HTML page. It has the following form:

<APPLET CODE="name of the applet class file" HEIGHT = maximum height of applet in pixels WIDTH = maximum width of applet in pixels ALIGN = alignment (LEFT, RIGHT, MIDDLE, TOP, BOTTOM)>
<PARAM NAME = parameter name VALUE = its value> </APPLET>

Execution: ***appletviewer programname.java or appletviewer programname.html***

The <PARAM> tag useful to define a variable (parameter) and its value inside the HTML page which can be passed to the applet. The applet can access the parameter value using **getParameter () method**, as: String value = getParameter ("pname");

Example Program:

Following is a simple applet named **HelloWorldApplet.java** −

```
import java.applet.*;
import java.awt.*;
public class HelloWorldApplet extends Applet {
  public void paint (Graphics g) {
    g.drawString ("Hello World", 25, 50);
  } }
```

Invoking an Applet - HelloWorldApplet.html

```
<html>
  <title>The Hello, World Applet</title>
    <applet code = "HelloWorldApplet.class" width = "320" height = "120">
  </applet>
</html>
```

OUTPUT: javac HelloWorldApplet.java
appletviewer HelloWorldApplet.html

Embedded an Applet

```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
/*
<applet code="First.class" width="300" height="300">
</applet>
*/
public class First extends Applet
{
  public void paint(Graphics g){
g.drawString("welcome to applet",150,150);
    }
  }
```

OUTPUT: javac First.java
appletviewer First.java

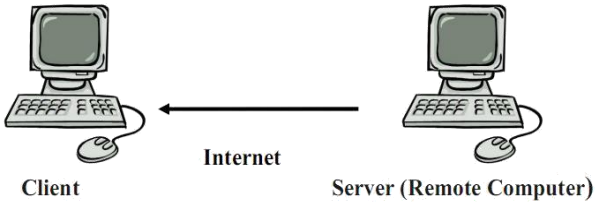
TYPES OF APPLETS

Applets are of two types:

// Local Applets
// Remote Applets

Local Applets: An applet developed locally and stored in a local system is called local applets. So, local system does not require internet. We can write our own applets and embed them into the web pages.

Remote Applets: The applet that is downloaded from a remote computer system and embed applet into a web page. The internet should be present in the system to download the applet and run it. To download the applet we must know the applet address on web known as Uniform Resource Locator(URL) and must be specified in the applets HTML document as the value of CODEBASE.



PASSING PARAMETERS TO AN APPLET

Java applet has the feature of retrieving the parameter values passed from the html page. So, you can pass the parameters from your html page to the applet embedded in your page. The **param** tag(**<parma name="" value=""></param**>) is used to pass the parameters to an applet. The applet has to call the getParameter() method supplied by the java.applet.Applet parent class.

Ex1: Write a program to pass employ name and id number to an applet.

```
import java.applet.*;
import java.awt.*;
/* <applet code="MyApplet2.class" width = 600 height= 450>
<param name = "t1" value="Hari Prasad"> <param name =
"t2" value ="101">
</applet> */

public class MyApplet2 extends Applet
{
String n;
String id;
public void init()
{
   n = getParameter("t1");
   id = getParameter("t2");
}
public void paint(Graphics g)
{
  g.drawString("Name is : " + n, 100,100);
  g.drawString("Id is : "+ id, 100,150);
}
}
```

Ex2: Write a program to pass two numbers and pass result to an applet.

```
import java.awt.*;
import java.applet.*;
  /*<APPLET code="Pp" width="300"
height="250"> <PARAM name="a" value="5">
<PARAM name="b" value="5">
</APPLET>*/
public class Pp extends Applet
{
      String str;
      int a,b,result;
public void init()
 {
           str=getParameter("a");
              a=Integer.parseInt(str);
          str=getParameter("b");
```

```
            b=Integer.parseInt(str);
          result=a+b;
          str=String.valueOf(result);
       }
  public void paint(Graphics g)
   {
        g.drawString(" Result of Addition is : "+str,0,15);
    }
}
```

Ex3: Hai.java

```
import java.applet.*;
import java.awt.*;
/*<Applet code="hai" height="250" width="250">
   <PARAM name="Message" value="Hai friend how are you ..?"></APPLET>
*/
 class hai extends Applet
  {
    private String defaultMessage = &amp;quot;Hello!&amp;quot;;
   public void paint(Graphics g) {

    String inputFromPage = this.getParameter(&amp;quot;Message&amp;quot;); if
    (inputFromPage == null) inputFromPage = defaultMessage;
    g.drawString(inputFromPage, 50, 55);
  }
}
```

Output:

EVENT HANDLING

Event handling is at the core of successful applet programming. Most events to which the applet will respond are generated by the user. The most commonly handled events are those generated by the mouse, the keyboard, and various controls, such as a push button.

Events are supported by the **java.awt.event** package.

The Delegation Event Model

The modern approach to handling events is based on the delegation event model, which defines standard and consistent mechanisms to generate and process events.

Its concept is quite simple: a source generates an event and sends it to one or more listeners. In this scheme, the listener simply waits until it receives an event. Once received, the listener processes the event and then returns.

The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to "delegate" the processing of an event to a separate piece of code.

In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.

EVENTS

In the delegation model, an event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.

Events may also occur that are not directly caused by interactions with a user interface.

For example, an event may be generated when a timer expires, a counter exceeds a value, software or hardware failure occurs, or an operation is completed.

EVENT SOURCES

A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method.

Here is the general form:

```
public void add Type Listener( Type Listener  el )
```

EVENT LISTENERS

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. The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**.

For example, the MouseMotionListener interface defines two methods to receive notifications when the mouse is dragged or moved.

EVENT CLASSES

The classes that represent events are at the core of Java's event handling mechanism. At the root of the Java event class hierarchy is EventObject, which is in **java.util**. It is the superclass for all events.

It's one constructor is shown here:
EventObject(Object src)

EventObject contains two methods: getSource( ) and toString( ) .

The **getSource( ) method** returns the source of the event. **Ex:** Object getSource( )
toString( ) returns the string equivalent of the event.

The package **java.awt.event** defines several types of events that are generated by various user interface elements.

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract super class for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; so occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved. (Added by Java 2, version 1.4)
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

The ActionEvent Class

An ActionEvent is generated when a button is pressed, a list item is double-clicked, or a menu item is selected. The ActionEvent class defines four integer constants that can be used to identify any modifiers associated with an action event: ALT_MASK , CTRL_MASK , META_MASK , and SHIFT_MASK .

ActionEvent has these three constructors:

ActionEvent(Object src , int type , String cmd)
ActionEvent(Object src , int type , String cmd , int modifiers)
ActionEvent(Object src , int type, String cmd, long when , int modifiers)

The ComponentEvent Class

// ComponentEvent is generated when the size, position, or visibility of a component is changed. There are four types of component events. The constants and their meanings are shown here:

COMPONENT_HIDDEN The component was hidden.
COMPONENT_MOVED The component was moved.
COMPONENT_RESIZED The component was resized.
COMPONENT_SHOWN The component became visible.

The ContainerEvent Class

//ContainerEvent is generated when a component is added to or removed from a container. There are two types of container events.

→ COMPONENT_ADDED and
COMPONENT_REMOVED **The KeyEvent Class**

//KeyEvent is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: KEY_PRESSED, KEY_RELEASED, and KEY_TYPED .

The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated.

The MouseEvent Class

There are **eight types of mouse events**. The MouseEvent class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED	The user clicked the mouse.
MOUSE_DRAGGED	The user dragged the mouse.
MOUSE_ENTERED	The mouse entered a component.
MOUSE_EXITED	The mouse exited from a component.
MOUSE_MOVED	The mouse moved.
MOUSE_PRESSED	The mouse was pressed.
MOUSE_RELEASED	The mouse was released.
MOUSE_WHEEL	The mouse wheel was moved (Java 2, v1.4).

The WindowEvent Class

There are **ten types of window events**. The WindowEvent class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	The window has been closed.
WINDOW_CLOSING	The user requested that the window be closed.
WINDOW_DEACTIVATED	The window was deactivated.
WINDOW_DEICONIFIED	The window was deiconified.
WINDOW_GAINED_FOCUS	The window gained input focus.
WINDOW_ICONIFIED	The window was iconified.
WINDOW_LOST_FOCUS	The window lost input focus.
WINDOW_OPENED	The window was opened.
WINDOW_STATE_CHANGED	The state of the window changed.

EVENT LISTENER INTERFACES

When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines **four methods** to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines **two methods** to recognize when a component is added to or removed from a container.
FocusListener	Defines **two methods** to recognize when a component gains or losses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines **three methods** to recognize when a key is pressed, released, or typed.
MouseListener	Defines **five methods** to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines **two methods** to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowListener	Defines **seven methods** to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

The delegation event model has two parts: sources and listeners. Listeners are created by implementing one or more of the interfaces defined by the **java.awt.event** package.

The ActionListener Interface

This interface defines the actionPerformed( ) method that is invoked when an action event occurs.

Its general form is shown here: void actionPerformed(ActionEvent ae)

The ItemListener Interface

This interface defines the itemStateChanged( ) method that is invoked when the state of an item changes.

Its general form is shown here: void itemStateChanged(ItemEvent ie)

The KeyListener Interface

This interface defines three methods. The keyPressed( ) and keyReleased( ) methods are invoked when a key is pressed and released, respectively. The keyTyped( ) method is invoked when a character has been entered. For example, if a user presses and releases the key, three events are generated in A sequence: key pressed, typed, and released.

The general forms of these methods are shown here:

void keyPressed(KeyEvent ke)
void keyReleased(KeyEvent ke)
void keyTyped(KeyEvent ke)

The MouseListener Interface

This interface defines five methods. If the mouse is pressed and released at the same point, mouseClicked( ) is invoked. When the mouse enters a component, the mouseEntered( ) method is called. When it leaves, mouseExited( ) is called. The mousePressed( ) and mouseReleased( ) methods are invoked when the mouse is pressed and released, respectively.

The general forms of these methods are shown here:

```
void mouseClicked(MouseEvent me )
void mouseEntered(MouseEvent me )
void mouseExited(MouseEvent  me )
void mousePressed(MouseEvent me )
void mouseReleased(MouseEvent me )
```

The MouseMotionListener Interface

This interface defines two methods. The mouseDragged( ) method is called multiple times as the mouse is dragged. The mouseMoved( ) method is called multiple times as the mouse is moved.

Their general forms are shown here:

```
void mouseDragged(MouseEvent me )
void mouseMoved(MouseEvent me )
```

The TextListener Interface

This interface defines the textChanged( ) method that is invoked when a change occurs in a text area or text field.

Its general form is shown here: void textChanged(TextEvent te)

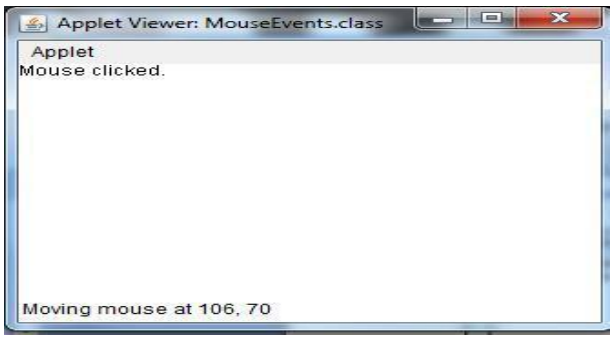
Handling Mouse Events

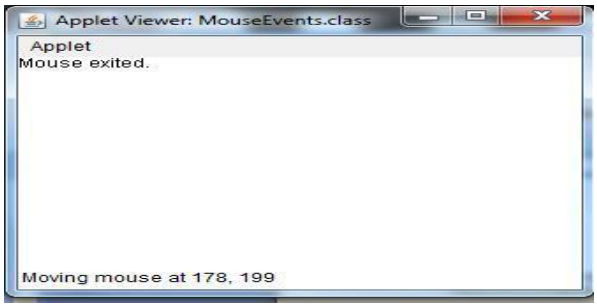
To handle mouse events, we must implement the MouseListener and the MouseMotion Listener interfaces.

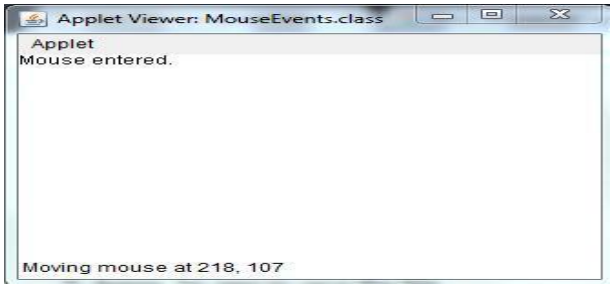
EX: // Demonstrate the mouse event handlers.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MouseEvents" width=300 height=100>
</applet>
*/
public class MouseEvents extends Applet implements MouseListener, MouseMotionListener
{ String msg = "";
int mouseX = 0, mouseY = 0; // coordinates of mouse
public void init() {
addMouseListener(this);
addMouseMotionListener(this);
}
```

```
// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
mouseX = 0;     // save coordinates
mouseY = 10;
msg = "Mouse clicked.";
repaint();
}

// Handle mouse entered.
public void mouseEntered(MouseEvent me) {
   //  save
      coordinates
      mouseX = 0;
      mouseY = 10;
msg = "Mouse entered.";
repaint();
}

   //  Handle mouse exited.
public void mouseExited(MouseEvent me) {
      //  save
         coordinate
         s mouseX
         = 0;
         mouseY =
         10;
msg = "Mouse exited.";
repaint();
}

      //  Handle button pressed.
public void mousePressed(MouseEvent me) {
//  save coordinates
   mouseX =
   me.getX();
   mouseY =
   me.getY(); msg =
   "Down"; repaint();
}

//  Handle button released.
public void mouseReleased(MouseEvent me) {
   1.  save
      coordinates
      mouseX =
      me.getX();
      mouseY =
      me.getY(); msg
      = "Up";
      repaint();
}
```







2. Handle mouse dragged.

```
public void mouseDragged(MouseEvent me) {
```

1. save coordinates

```
   mouseX =
   me.getX();
```

```
mouseY = me.getY();
msg = "*";
showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
repaint();
}

// Handle mouse moved.
public void mouseMoved(MouseEvent me) {
// show status
showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
}

//  Display msg in applet window at current X,Y location.
  public void paint(Graphics g) {
g.drawString(msg, mouseX, mouseY);
}
}
```

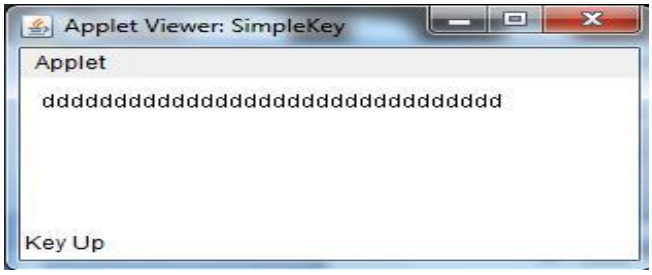
Handling Keyboard Events

When a key is pressed, a **KEY_PRESSED** event is generated. This results in a call to the keyPressed( ) event handler. When the key is released, a **KEY_RELEASED** event is generated and the keyReleased( ) handler is executed. If a character is generated by the keystroke, then a **KEY_TYPED** event is sent and the keyTyped( ) handler is invoked.

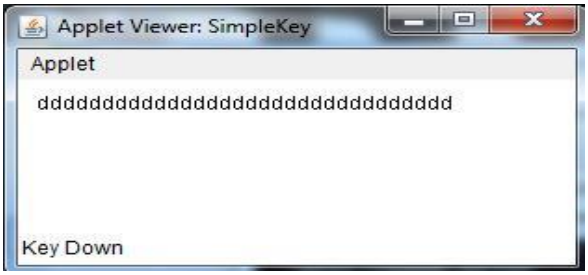
Thus, each time the user presses a key, at least two and often three events are generated. If all you care about are actual characters, then you can ignore the information passed by the key press and release events.

EX:

```
// Demonstrate the key event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/* <applet code="SimpleKey" width=300
height=100> </applet> */
public class SimpleKey extends Applet implements KeyListener
  {
    String msg = "";
    int X = 10, Y = 20; // output coordinates
    public void init() {
    addKeyListener(this);
    requestFocus(); // request input focus
}

 public void keyPressed(KeyEvent ke)
{ showStatus("Key Down"); }

 public void keyReleased(KeyEvent ke) {
showStatus("Key Up");
}
```



```
public void keyTyped(KeyEvent ke) {
msg += ke.getKeyChar();
repaint();
}

// Display keystrokes.
public void paint(Graphics g)
{ g.drawString(msg, X, Y); }

}
```



Adapter Classes

Java provides a special feature, called an adapter class , that can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface.

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

For example, the MouseMotionAdapter class has two methods, mouseDragged( ) and mouseMoved( ) . The signatures of these empty methods are exactly as defined in the MouseMotionListener interface. If you were interested in only mouse drag events, then you could simply extend MouseMotionAdapter and implement mouseDragged( ) . The empty implementation of mouseMoved( ) would handle the mouse motion events for you.

Adapter Class	**Listener Interface**
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

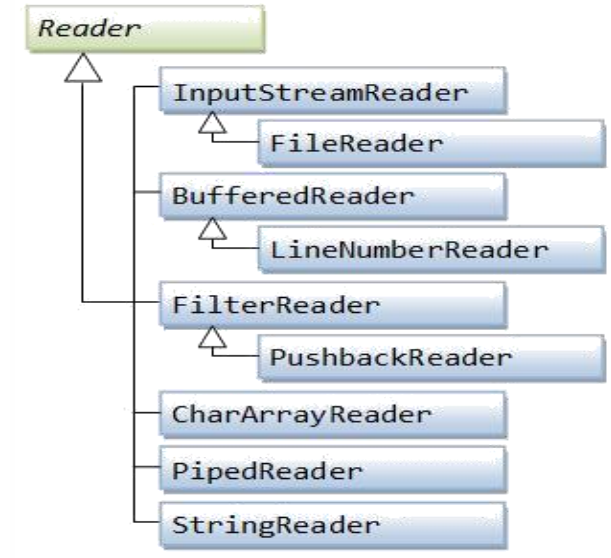
EX: // Demonstrate an adapter.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300 height=100>
</applet>
*/

public class AdapterDemo extends Applet
{
public void init() {
addMouseListener(new MyMouseAdapter(this));
addMouseMotionListener(new
MyMouseMotionAdapter(this)); }
}
```

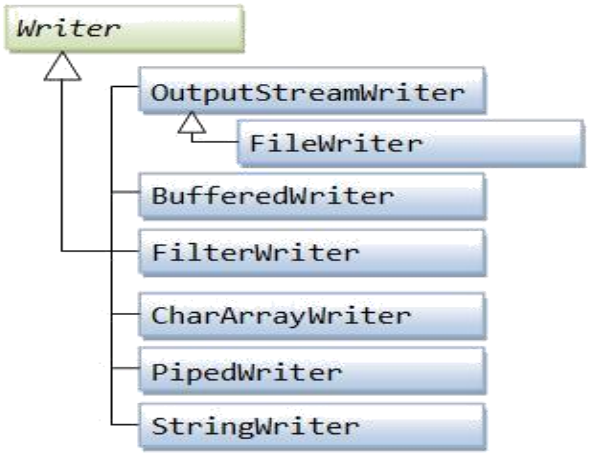
```
class MyMouseAdapter extends MouseAdapter
{
AdapterDemo adapterDemo;
public MyMouseAdapter(AdapterDemo adapterDemo)
{
  this.adapterDemo = adapterDemo;
}

// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
adapterDemo.showStatus("Mouse clicked");
}
}

class MyMouseMotionAdapter extends
 MouseMotionAdapter {
  AdapterDemo adapterDemo;
  public MyMouseMotionAdapter(AdapterDemo adapterDemo)
   {
      this.adapterDemo = adapterDemo;
    }

// Handle mouse dragged.
public void mouseDragged(MouseEvent me)
 {
   adapterDemo.showStatus("Mouse dragged");
  }
 }
```

Files and Streams:

Text stream classes for reading data



Text stream classes for writing data



Stream

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

- **InPutStream** − The InputStream is used to read data from a source.
- **OutPutStream** − The OutputStream is used for writing data to a destination.

Java provides strong but flexible support for I/O related to files and networks.

Byte Streams

Java byte streams are used to perform **input and output of 8-bit bytes**. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**.

Example

```
import java.io.*;
public class CopyFile {
  public static void main(String args[]) throws IOException {
    FileInputStream in = null;
    FileOutputStream out = null;
    try {
      in = new FileInputStream("input.txt");
      out = new FileOutputStream("output.txt");
      int c;
      while ((c = in.read()) != -1) {
        out.write(c);
      }
    }finally {
      if (in != null) {
        in.close();
      }
      if (out != null) {
        out.close();
      }     }    } }
```

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

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

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

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform **input and output for 16-bit unicode**. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**.

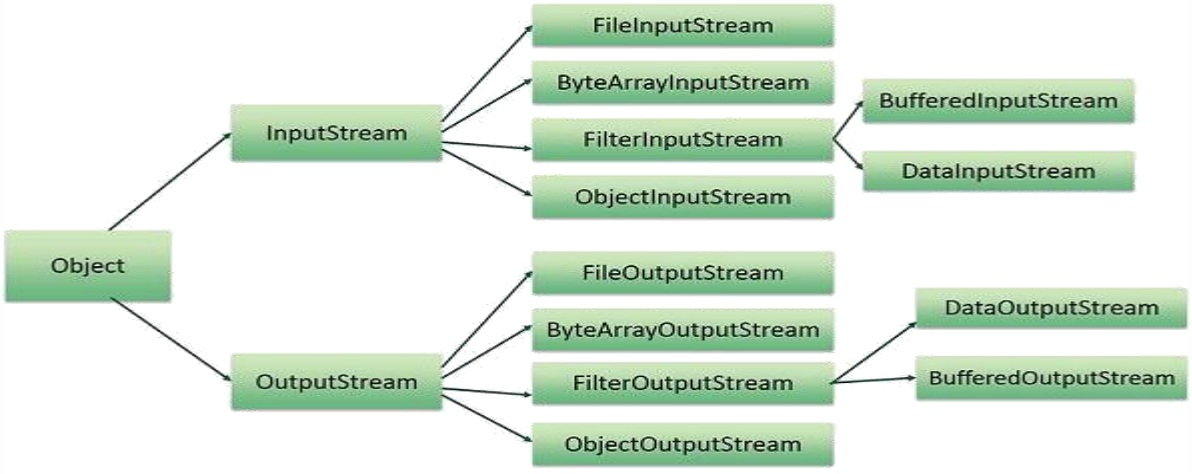
Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

Example

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

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

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

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

Reading and Writing Files Text input/output,

As described earlier, a stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination. Here is a hierarchy of classes to deal with Input and Output streams.



The two important streams are **FileInputStream** and **FileOutputStream** (File Handling)

In Java, FileInputStream and FileOutputStream classes are used to read and write data in file. In another words, they are used for file handling in java.

Java FileOutputStream class

Java FileOutputStream is an output stream for writing data to a file. It is a class belongs to **byte streams.** It can be used to create text files.

→ First we should read data from the keyword. It uses **DataInputStream** class for reading data from the keyboard is as:

DataInputStream dis=new DataInputStream(System.in);

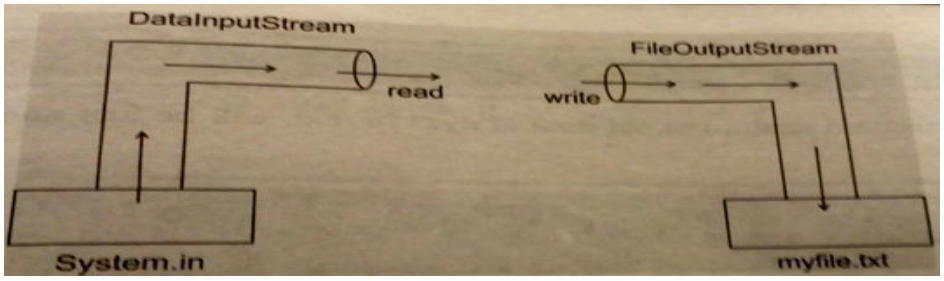
→ FileOutputStream **used to send data to the file and attaching the file to FileOutputStream.** i.e., **FileOutputStream fout=new FileOutputStream("File_name");**

→ The next step is to read data from DataInputStream and write it into FileOutputStream. It means read data from **dis object** and write it into **fout object. i.e.,**

```
ch=(char)dis.read();  //read one character into ch
fout.write(ch);   //write ch into file.
```

→ **Finally** closing the file using: **fout.close();**

Creating a Text file:



Example: Write a program to read data from the keyboard and write it to myfile.txt file.

```
import java.io.*;
class Test{
 public static void main(String args[])
  {
   DataInputStream dis=new DataInputStream(System.in);
   FileOutputstream fout=new
   FileOutputStream("myfile.txt"); System.out.println("Enter
   text @ at the end:"); char ch;
    while((ch=(char)dis.read())!="@")
     fout.write(ch);
     fout.close();
 }
}
```

Output: javac Test.java
Java Test

```
Enter text @ at the end:
This is my file line one
This is my file line two@
```

Java FileInputStream class

It is useful to read data from a file in the form of sequence of bytes. It is possible to read data from a text file using FileInputStream. i.e.,

FileInputStream fin= new FileInputStream("myfile.txt");

To read data from the file is, **ch=fin.read();**

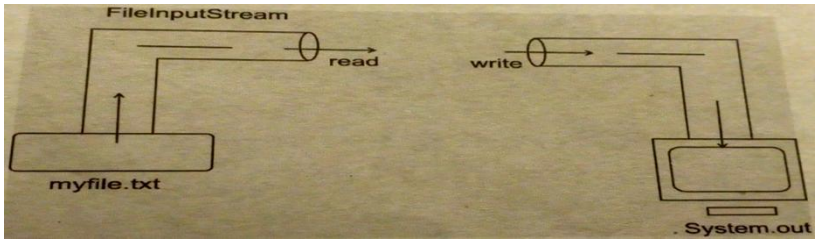
When there is no more data available to read then it returns **-1**.

→ The Output Stream is used to send data to the **monitor**. i.e., **PrintStream,** for displaying the data we can use **System.out.**

System.out.print(ch);

Reading data from a text file using FileInputStream:

Java FileInputStream class obtains input bytes from a file. It is used for reading streams of raw bytes such as image data. It should be used to read byte-oriented data for example to read image, audio, video etc.



Example: Write a program to read data from myfile.txt using FileInputStream and display it on monitor.

```
import java.io.*;
class ReadFile
{
 public static void main(String args[])
  {

   FileInputStream fin=new FileInputStream("myfile.txt");
    System.out.println("File Contents:"); int ch;
   while((ch=fin.read())!=-1)
   {
    System.out.println((char)ch);
   }
   fin.close();
  }
 }
```

Output: javac ReadFile.java
java ReadFile

File Contents:
This is my file line one
This is my file line two

UNIT V: GUI Programming with Java – AWT class hierarchy, component, container, panel, window, frame, graphics.
AWT controls: Labels, button, text field, check box, check box groups, choices, lists, scrollbars, and graphics.
Layout Manager – Layout manager types: border, grid and flow.
Swing – Introduction, limitations of AWT, Swing vs AWT.

GUI PROGRAMMING WITH JAVA

ABSTRACT WINDOW TOOLKIT (AWT)

Java AWT (Abstract Window Toolkit) is *an **API to develop GUI** or window-based application in java*. Java AWT components are **platform-dependent** i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components uses the resources of system. The Abstract Window Toolkit(AWT) support for applets. The AWT contains numerous classes and methods that allow you to create and manage windows.

The **java.awt package** provides classes for AWT api such as **TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.**

AWT Classes
The AWT classes are contained in the **java.awt package**. It is one of Java's largest packages.

Class	Description
AWTEvent	Encapsulates AWT events.
AWTEventMulticaster	Dispatches events to multiple listeners.
BorderLayout	Border layouts use five components: North, South, East, West, and Center.
CardLayout	Card layouts emulate index cards. Only the one on top is showing.
Checkbox	Creates a check box control.
CheckboxGroup	Creates a group of check box controls.
CheckboxMenuItem	Creates an on/off menu item.
Choice	Creates a pop-up list.
Color	Manages colors in a portable, platform-independent fashion.
Component	An abstract superclass for various AWT components.
Container	A subclass of Component that can hold other components.
Cursor	Encapsulates a bitmapped cursor.
Dialog	Creates a dialog window.
Dimension	Specifies the dimensions of an object. The width is stored in width , and the height is stored in height .
Event	Encapsulates events.
FlowLayout	The flow layout manager. Flow layout positions components left to right, top to bottom.

Frame	Creates a standard window that has a title bar, resize corners, and a menu bar.
Graphics	Encapsulates the graphics context.

Control Fundamentals

The AWT supports the following types of controls:

3. Labels
4. Push buttons
5. Check boxes
6. Choice lists
7. Lists
8. Scroll bars
9. Text editing

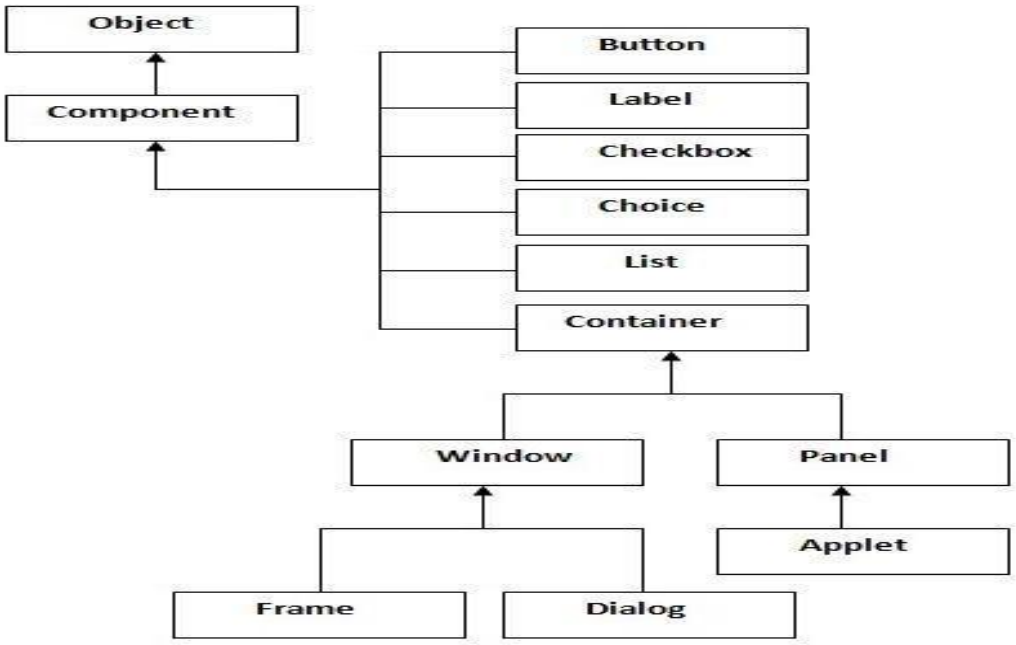
User interaction with the program is of two types:

CUI (Character User Interface): In CUI user interacts with the application by typing characters or commands. In CUI user should remember the commands. It is not user friendly.

2. **GUI (Graphical User Interface):** In GUI user interacts with the application through graphics. GUI is user friendly. GUI makes application attractive. It is possible to simulate real object in GUI programs. In java to write GUI programs we can use awt (Abstract Window Toolkit) package.

Java AWT Class Hierarchy

The hierarchy of Java AWT classes is given below.



Container

The Container is a component in AWT that can contain other components like buttons, textfields, labels etc. The classes that extend Container class are known as container such as **Frame, Dialog and Panel**.

Window

The window is the container that has no borders and menu bars. You must use frame, dialog or another window for creating a window.

Panel

The Panel is the container that doesn't contain title bar and menu bars. It can have other components like button, textfield etc.

Frame

The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

Useful Methods of Component class

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.

Listeners and Listener Methods:

Listeners are available for components. A Listener is an interface that listens to an event from a component. Listeners are available in **java.awt.event package**. The methods in the listener interface are to be implemented, when using that listener.

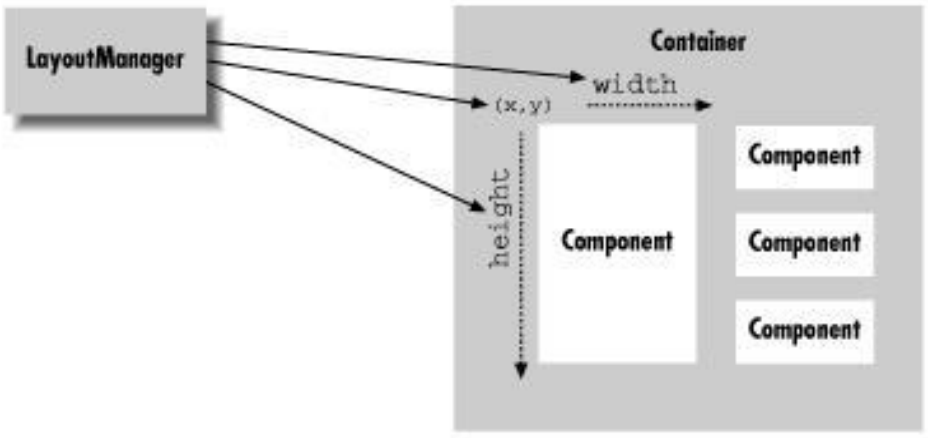
Component	Listener	Listener methods
Button	ActionListener	public void actionPerformed (ActionEvent e)
Checkbox	ItemListener	public void itemStateChanged (ItemEvent e)
CheckboxGroup	ItemListener	public void itemStateChanged (ItemEvent e)
TextField	ActionListener FocusListener	public void actionPerformed (ActionEvent e) public void focusGained (FocusEvent e) public void focusLost (FocusEvent e)
TextArea	ActionListener FocusListener	public void actionPerformed (ActionEvent e) public void focusGained (FocusEvent e) public void focusLost (FocusEvent e)
Choice	ActionListener ItemListener	public void actionPerformed (ActionEvent e) public void itemStateChanged (ItemEvent e)
List	ActionListener ItemListener	public void actionPerformed (ActionEvent e) public void itemStateChanged (ItemEvent e)
Scrollbar	AdjustmentListener MouseMotionListener	public void adjustmentValueChanged (AdjustmentEvent e) public void mouseDragged (MouseEvent e) public void mouseMoved (MouseEvent e)
Label	No listener is needed	

Layout Managers

A layout manager arranges the child components of a container. It positions and sets the size of components within the container's display area according to a particular layout scheme.

The layout manager's job is to fit the components into the available area, while maintaining the proper spatial relationships between the components. AWT comes with a few standard layout managers that will collectively handle most situations; you can make your own layout managers if you have special requirements.

LayoutManager at work



Every container has a **default layout manager**; therefore, when you make a new container, it comes with a LayoutManager object of the appropriate type. You can install a new layout manager at any time with the setLayout() method. Below, we set the layout manager of a container to a BorderLayout:

setLayout (new BorderLayout( ));

Every component determines three important pieces of information used by the layout manager in placing and sizing it: a minimum size, a maximum size, and a preferred size.

These are reported by the getMinimumSize(), getMaximumSize(), and getPreferredSize(), methods of Component, respectively.

When a layout manager is called to arrange its components, it is working within a fixed area. It usually begins by looking at its container's dimensions, and the preferred or minimum sizes of the child components.

Layout manager types

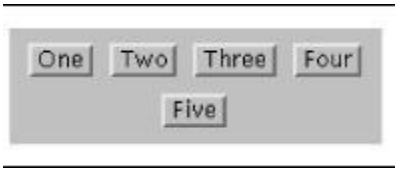
Flow Layout

`FlowLayout` is a simple layout manager that tries to arrange components with their preferred sizes, from left to right and top to bottom in the display. A `FlowLayout` can have a specified justification of `LEFT`, `CENTER`, or `RIGHT`, and a fixed horizontal and vertical padding.

By default, a flow layout uses **`CENTER`** justification, meaning that all components are centered within the area allotted to them. `FlowLayout` is the default for `Panel` components like `Applet`.

The following applet adds five buttons to the default `FlowLayout`.

```
import java.awt.*;
/*
<applet code="Flow" width="500" height="500">
</applet>
*/
public class Flow extends java.applet.Applet
{
   public void init()
  {
        //Default for Applet is FlowLayout
             add( new Button("One") );
       add( new Button("Two") );
       add( new Button("Three") );
       add( new Button("Four") );
       add( new Button("Five") );
   }
}
```



If the applet is small enough, some of the buttons spill over to a second or third row.

Grid Layout

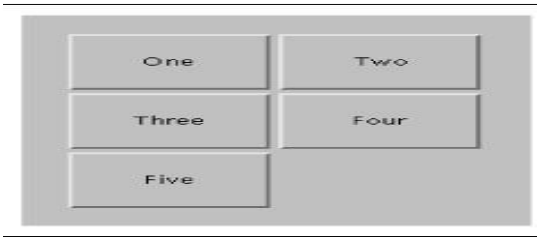
`GridLayout` arranges components into regularly spaced rows and columns. The components are arbitrarily resized to fit in the resulting areas; their minimum and preferred sizes are consequently ignored.

`GridLayout` is most useful for arranging very regular, identically sized objects and for allocating space for Panels to hold other layouts in each region of the container.

`GridLayout` takes the number of rows and columns in its constructor. If you subsequently give it too many objects to manage, it adds extra columns to make the objects fit. You can also set the number of rows or columns to zero, which means that you don't care how many elements the layout manager packs in that dimension.

For example, `GridLayout(2,0)` requests a layout with two rows and an unlimited number of columns; if you put ten components into this layout, you'll get two rows of five columns each. The following applet sets a `GridLayout` with three rows and two columns as its layout manager;

```
import java.awt.*;
/*
   <applet code="Grid" width="500"
    height="500"> </applet>
*/
public class Grid extends java.applet.Applet
  {
   public void init()
    {
      setLayout( new GridLayout( 3, 2 ));
```



```
      add( new Button("One") );
      add( new Button("Two") );
      add( new Button("Three") );
      add( new Button("Four") );
      add( new Button("Five") );
   }
}
```

The five buttons are laid out, in order, from left to right, top to bottom, with one empty spot.

Border Layout

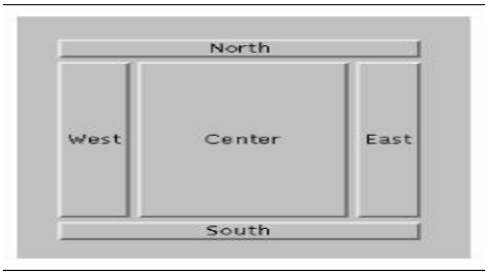
`BorderLayout` is a little more interesting. It tries to arrange objects in one of five geographical locations: "North," "South," "East," "West," and "Center," possibly with some padding between.

`BorderLayout` is the default layout for `Window` and `Frame` objects. Because each component is associated with a direction, `BorderLayout` can manage at most five components; it squashes or stretches those components to fit its constraints.

When we add a component to a border layout, we need to specify both the component and the position at which to add it. To do so, we use an overloaded version of the **`add()`** method that takes an additional argument as a constraint.

The following applet sets a `BorderLayout` layout and adds our five buttons again, named for their locations;

```
import java.awt.*;
/*
<applet code="Border" width="500" height="500">
</applet>
*/
public class Border extends java.applet.Applet
  {
    public void init()
   {
      setLayout( new java.awt.BorderLayout() );
      add( new Button("North"), "North" ); add(
      new Button("East"), "East" );
      add( new Button("South"), "South" );
      add( new Button("West"), "West" );
      add( new Button("Center"), "Center" );
   }
}
```



Compile: javac Border.java
Run : appletviewer Border.java

Java AWT Example

To create simple awt example, you need a frame. There are two ways to create a frame in AWT.

// By extending Frame class (inheritance)
2. By creating the object of Frame class (association)

Simple example of AWT by inheritance

```
import java.awt.*;
class First extends Frame{
First(){
Button b=new Button("click me");
b.setBounds(30,100,80,30);// setting button position
add(b);//adding button into frame
setSize(300,300);//frame size 300 width and 300
height setLayout(null);//no layout manager
setVisible(true);//now frame will be visible }

public static void main(String args[]){
First f=new First();
}
}
```

Simple example of AWT by association

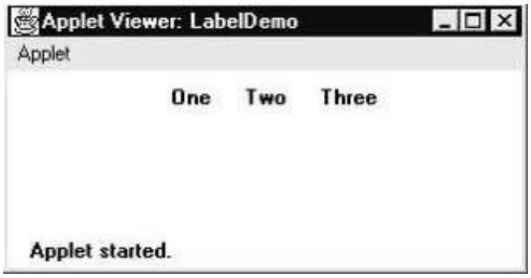
```
import java.awt.*;
class First2{
First2(){
Frame f=new Frame();
 Button b=new Button("click me");
b.setBounds(30,50,80,30);
 f.add(b);
f.setSize(300,300);
f.setLayout(null);
f.setVisible(true);
}
public static void main(String args[]){
First2 f=new First2();
}
}
```

AWT controls

Labels:

The easiest control to use is a label. A label is an object of type Label, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user.

```
 // Demonstrate
    Labels import
    java.awt.*; import
    java.applet.*; /*
<applet code="LabelDemo" width=300 height=200>
</applet> */
public class LabelDemo extends Applet
{
 public void init()
 {
   Label one = new Label("One");
   Label two = new Label("Two");
   Label three = new Label("Three");
        add labels to applet window
  add(one);
  add(two);
  add(three);
}
}
```



Buttons:

The most widely used control is the push button. 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.

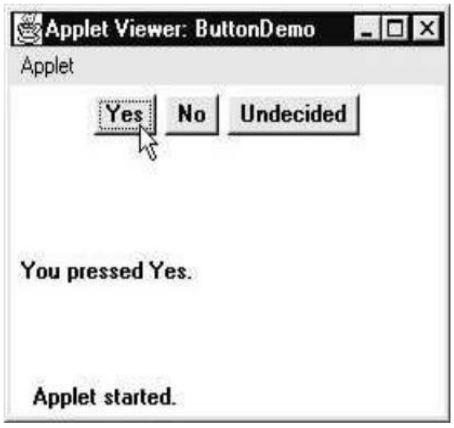
Button class is useful to create push buttons. A push button triggers a series of events.

To create push button: Button b1 =new Button("label");
To get the label of the button: String l = b1.getLabel();
To set the label of the button: b1.setLabel("label");
To get the label of the button clicked: String str = ae.getActionCommand();

-

```
\{ Demonstrate Buttons
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/* <applet code="ButtonDemo" width=250 height=150>
</applet> */

public class ButtonDemo extends Applet implements ActionListener
  {
String msg = "";
Button yes, no, maybe;
```

```
public void init()
{
yes = new Button("Yes");
no = new Button("No");
maybe = new Button("Undecided");
add(yes);
add(no);
add(maybe);
yes.addActionListener(this);
no.addActionListener(this);
maybe.addActionListener(this);
}

public void actionPerformed(ActionEvent ae)
{
String str = ae.getActionCommand();
  if(str.equals("Yes"))
    {
      msg = "You pressed Yes.";
     }
else if(str.equals("No"))
  {
    msg = "You pressed No.";
   }
else
 {
    msg = "You pressed Undecided.";
  }
 repaint();
}
public void paint(Graphics g)
{
   g.drawString(msg, 6, 100);
}
  }
```



Check Boxes:

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.

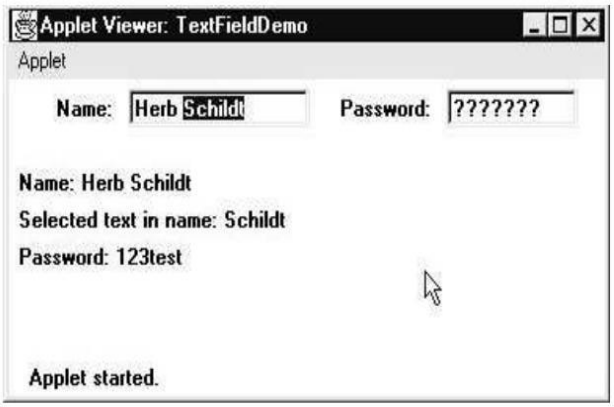
```
\{    Demonstrate check boxes.
   import java.awt.*;
    import java.awt.event.*;
    import java.applet.*;
```

```
/*
<applet code="CheckboxDemo" width=250 height=200>
</applet>
*/
 public class CheckboxDemo extends Applet implements ItemListener
{
String msg = "";
checkbox Win98, winNT, solaris, mac;

public void init()
 {
  win98 = new Checkbox("Windows 98/XP", null, true);
  winNT = new Checkbox("Windows NT/2000");
  solaris = new Checkbox("Solaris"); mac = new
  Checkbox("MacOS");
add(Win98);
add(winNT);
add(solaris);
add(mac);
Win98.addItemListener(this);
winNT.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie)
{
 repaint();
}
// Display current state of the check boxes.

public void paint(Graphics g)
{
  msg = "Current state: ";
   g.drawString(msg, 6, 80);
  msg = " Windows 98/XP: " + Win98.getState();
   g.drawString(msg, 6, 100);
  msg = " Windows NT/2000: " + winNT.getState();
   g.drawString(msg, 6, 120);
  msg = " Solaris: " + solaris.getState();
   g.drawString(msg, 6, 140);
  msg = " MacOS: " + mac.getState();
   g.drawString(msg, 6, 160);
 }
}
```

TextField:

The TextField class implements a single-line text-entry area, usually called an edit control. Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections.

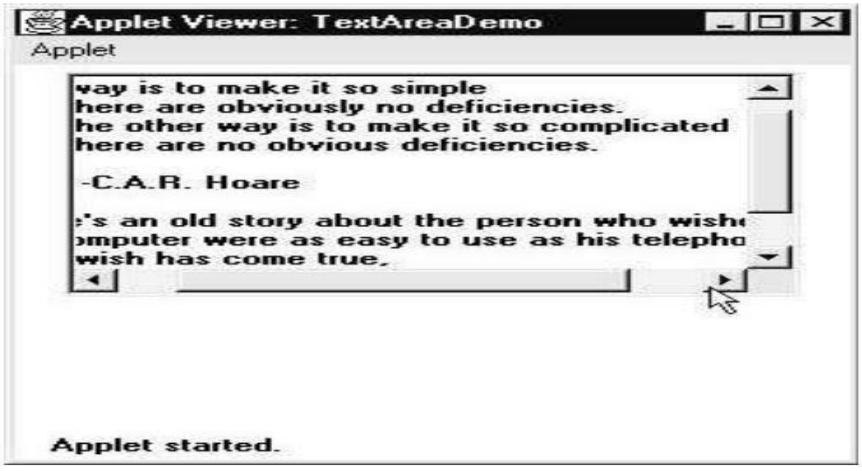
```
//  Demonstrate text field.
    import java.awt.*;
    import
    java.awt.event.*;
    import java.applet.*;
/*
<applet code="TextFieldDemo" width=380 height=150>
</applet>
*/
 public class TextFieldDemo extends Applet implements ActionListener
 {
   TextField name, pass;

 public void init()
 {
 Label namep = new Label("Name: ", Label.RIGHT);
 Label passp = new Label("Password: ", Label.RIGHT);
 name = new TextField(12);
 pass = new TextField(8);
 pass.setEchoChar('?');
 add(namep);
 add(name);
 add(passp);
 add(pass);
//  register to receive action events
     name.addActionListener(this);
     pass.addActionListener(this);
 }
//  User pressed Enter.
 public void actionPerformed(ActionEvent ae)
 {
 repaint();
 }
 public void paint(Graphics g)
 {
  g.drawString("Name: " + name.getText(), 6, 60);
  g.drawString("Selected text in name: " + name.getSelectedText(), 6, 80);
  g.drawString("Password: " + pass.getText(), 6, 100);
 }
 }
```



TextArea:

Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called TextArea .

```
\{  Demonstrate
    TextArea. import
    java.awt.*; import
    java.applet.*;
/*
<applet code="TextAreaDemo" width=300
height=250> </applet>
*/
public class TextAreaDemo extends Applet
{
  public void init()
  {
String val = "There are two ways of constructing " + "a software design.\n" + "One way is to make it so simple\n" + "that there are obviously no deficiencies.\n" + "And the other way is to make it so complicated\n" + "that there are no obvious deficiencies.\n\n" + " -C.A.R. Hoare\n\n" + "There's an old story about the person who wished\n" + "his computer were as easy to use as his telephone.\n" + "That wish has come true,\n" + "since I no longer know how to use my telephone.\n\n" + " -Bjarne Stroustrup, AT&T, (inventor of C++)";

TextArea text = new TextArea(val, 10, 30);
add(text);
}
}
```



CheckboxGroup

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called radio buttons. A Radio button represents a round shaped button such that only one can be selected from a panel. Radio button can be created using CheckboxGroup class and Checkbox classes.

· To create a radio button: CheckboxGroup cbg = new CheckboxGroup ();
Checkbox cb = new Checkbox ("label", cbg, true);
· To know the selected checkbox: Checkbox cb = cbg.getSelectedCheckbox ();
·To know the selected checkbox label: String label = cbg.getSelectedCheckbox().getLabel ();

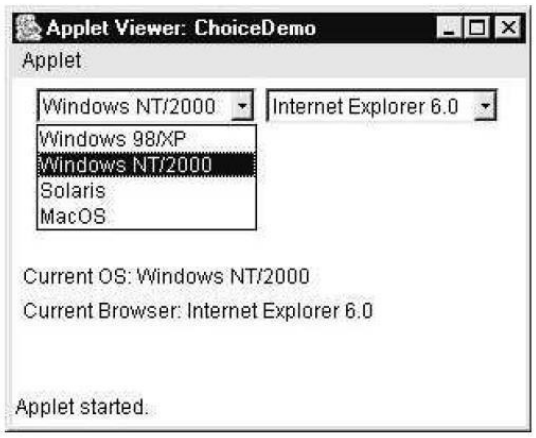
```
\{  Demonstrate check box group.
 import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/*
<applet code="CBGroup" width=250 height=200>
</applet>
*/

public class CBGroup extends Applet implements ItemListener
{
String msg = "";
Checkbox Win98, winNT, solaris, mac;
CheckboxGroup cbg;
public void init() {
cbg = new CheckboxGroup();
Win98 = new Checkbox("Windows 98/XP", cbg, true);
winNT = new Checkbox("Windows NT/2000", cbg, false);
solaris = new Checkbox("Solaris", cbg, false); mac = new
Checkbox("MacOS", cbg, false);
add(Win98);
add(winNT);
add(solaris);
add(mac);
Win98.addItemListener(this);
winNT.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie) {
repaint();
}
 \{ Display current state of the check
      boxes. public void paint(Graphics g) {
msg = "Current selection: ";
msg += cbg.getSelectedCheckbox().getLabel();
g.drawString(msg, 6, 100);
}
}
```

Choice Controls

The Choice class is used to create a pop-up list of items from which the user may choose. Thus, a Choice control is a form of menu. Choice menu is a popdown list of items. Only one item can be selected.

· To create a choice menu: Choice ch = new Choice();
· To add items to the choice menu: ch.add ("text");
· To know the name of the item selected from the choice menu:
String s = ch.getSelectedItem ();
· To know the index of the currently selected item: int i = ch.getSelectedIndex();
This method returns -1, if nothing is selected.

```
  //Demonstrate Choice
      lists. import
      java.awt.*; import
      java.awt.event.*;
      import java.applet.*;
/*
<applet code="ChoiceDemo" width=300
height=180> </applet>
*/
public class ChoiceDemo extends Applet implements ItemListener
{ Choice os, browser;
String msg = ""; public
void init() { os = new
Choice(); browser = new
Choice();
  //add items to os list
os.add("Windows 98/XP");
os.add("Windows NT/2000");
os.add("Solaris");
os.add("MacOS");
  \{  add items to browser list
      browser.add("Netscape 3.x");
      browser.add("Netscape 4.x");
      browser.add("Netscape 5.x");
      browser.add("Netscape 6.x");
      browser.add("Internet Explorer
      4.0"); browser.add("Internet
      Explorer 5.0");
      browser.add("Internet Explorer
      6.0"); browser.add("Lynx 2.4");
      browser.select("Netscape 4.x");
  \{  add choice lists to
      window add(os);
add(browser);
  \{  register to receive item
      events
      os.addItemListener(this);
      browser.addItemListener(
      this);
}
```



```
public void itemStateChanged(ItemEvent ie) {
repaint();
}
// Display current selections.
public void paint(Graphics g)
{ msg = "Current OS: ";
msg += os.getSelectedItem();
g.drawString(msg, 6, 120);
msg = "Current Browser: ";
msg += browser.getSelectedItem();
g.drawString(msg, 6, 140);
}
}
```

Lists

The List class provides a compact, multiple-choice, scrolling selection list. Unlike the Choice object, which shows only the single selected item in the menu, a List object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections. List provides these constructors: List( )

List(int numRows)

List(int numRows , boolean multipleSelect)

A List box is similar to a choice box, it allows the user to select multiple items.

· To create a list box: List lst = new List();

(or)

List lst = new List (3, true);

This list box initially displays 3 items. The next parameter true represents that the user can select more than one item from the available items. If it is false, then the user can select only one item.

= To add items to the list box: lst.add("text");
= To get the selected items: String x[] = lst.getSelectedItems();
= To get the selected indexes: int x[] = lst.getSelectedIndexes ();

```
// Demonstrate Lists.
import java.awt.*; import
java.awt.event.*; import
java.applet.*; /*
<applet code="ListDemo" width=300 height=180>
</applet>
*/
public class ListDemo extends Applet implements ActionListener
{ List os, browser;
String msg = "";
public void init() { os
= new List(4, true);
```

```
  browser = new List(4, false);
\}  add items to os list
    os.add("Windows 98/XP");
    os.add("Windows
    NT/2000");
    os.add("Solaris");
    os.add("MacOS");
\}  add items to browser list
    browser.add("Netscape 3.x");
    browser.add("Netscape 4.x");
    browser.add("Netscape 5.x");
    browser.add("Netscape 6.x");
    browser.add("Internet Explorer
    4.0"); browser.add("Internet
    Explorer 5.0");
    browser.add("Internet Explorer
    6.0"); browser.add("Lynx 2.4");
    browser.select(1);
\}  add lists to window
  add(os);
  add(browser);
//   register to receive action events
    os.addActionListener(this);
    browser.addActionListener(this);
  }
  public void actionPerformed(ActionEvent ae)
  { repaint();
  }
//   Display current selections.
  public void paint(Graphics g)
  {
  int idx[];
  msg = "Current OS: ";
  idx = os.getSelectedIndexes();
  for(int i=0; i<idx.length; i++)
  msg += os.getItem(idx[i]) + " ";
  g.drawString(msg, 6, 120);
  msg = "Current Browser: ";
  msg += browser.getSelectedItem();
  g.drawString(msg, 6, 140);
  }
  }
```

Scroll Bars

Scroll bars are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. Scrollbar class is useful to create scrollbars that can be attached to a frame or text area. Scrollbars can be arranged vertically or horizontally.

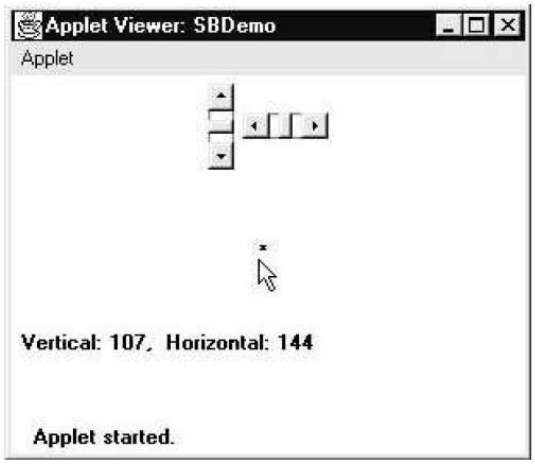
\{ To create a scrollbar : Scrollbar sb = new Scrollbar (alignment, start, step, min, max);
alignment: Scrollbar.VERTICAL, Scrollbar.HORIZONTAL
start: starting value (e.g. 0)
step: step value (e.g. 30) // represents scrollbar length
min: minimum value (e.g. 0)
max: maximum value (e.g. 300)

· To know the location of a scrollbar: int n = sb.getValue ();
· To update scrollbar position to a new position: sb.setValue (int position);
· To get the maximum value of the scrollbar: int x = sb.getMaximum ();
· To get the minimum value of the scrollbar: int x = sb.getMinimum ();
· To get the alignment of the scrollbar: int x = getOrientation ();

This method return 0 if the scrollbar is aligned HORIZONTAL, 1 if aligned VERTICAL.

```
// Demonstrate scroll bars.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SBDemo" width=300 height=200>
</applet>
*/
public class SBDemo extends Applet
implements AdjustmentListener, MouseMotionListener
{ String msg = "";
Scrollbar vertSB, horzSB;
public void init() {
int width = Integer.parseInt(getParameter("width")); int
height = Integer.parseInt(getParameter("height"));
vertSB = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 0, height);
horzSB = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, width);
add(vertSB);
add(horzSB);
// register to receive adjustment events
vertSB.addAdjustmentListener(this);
horzSB.addAdjustmentListener(this);
addMouseMotionListener(this);
}
public void adjustmentValueChanged(AdjustmentEvent ae) {
repaint();
}
// Update scroll bars to reflect mouse dragging.
public void mouseDragged(MouseEvent me) {
int x = me.getX();
int y = me.getY();
vertSB.setValue(y);
horzSB.setValue(x);
```

```
    repaint();
    }
//    Necessary for MouseMotionListener
  public void mouseMoved(MouseEvent me) {
    }
//    Display current value of scroll bars.
    public void paint(Graphics g) {
    msg = "Vertical: " + vertSB.getValue();
    msg += ", Horizontal: " + horzSB.getValue();
    g.drawString(msg, 6, 160);
  \}  show current mouse drag position
    g.drawString("*", horzSB.getValue(),
    vertSB.getValue());
    }
    }
```



Graphics

The AWT supports a rich assortment of graphics methods. All graphics are drawn relative to a window.

Graphics class and is 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.

Drawing Lines

Lines are drawn by means of the drawLine( ) method, shown here:

void drawLine(int startX, int startY, int endX, int endY)

drawLine( ) displays a line in the current drawing color that begins at startX,startY and ends at endX,endY.

The following applet draws several lines:

```
//    Draw lines import
  java.awt.*; import
  java.applet.*; /*

   <applet code="Lines" width=300
   height=200> </applet>
   */
   public class Lines extends Applet
   { public void paint(Graphics g) {
   g.drawLine(0, 0, 100, 100);
   g.drawLine(0, 100, 100, 0);
   g.drawLine(40, 25, 250, 180);
   g.drawLine(75, 90, 400, 400);
   g.drawLine(20, 150, 400, 40);
   g.drawLine(5, 290, 80, 19);
   }  }
```

Drawing Rectangles

The drawRect( ) and fillRect( ) methods display an outlined and filled rectangle, respectively. They are shown here:

void drawRect(int top, int left, int width, int height)
void fillRect(int top, int left, int width, int height)

The upper-left corner of the rectangle is at top,left. The dimensions of the rectangle are specified by width and height.

To draw a rounded rectangle, use drawRoundRect( ) or fillRoundRect( ), both shown here: void drawRoundRect(int top, int left, int width, int height,int xDiam, int yDiam)
void fillRoundRect(int top, int left, int width, int height, int xDiam, int yDiam)

```
// Draw rectangles
import java.awt.*;
import java.applet.*;
/*
<applet code="Rectangles" width=300
height=200> </applet>
*/
public class Rectangles extends Applet {
public void paint(Graphics g) {
g.drawRect(10, 10, 60, 50); g.fillRect(100,
10, 60, 50); g.drawRoundRect(190, 10,
60, 50, 15, 15); g.fillRoundRect(70, 90,
140, 100, 30, 40);
}
}
```

Drawing Ellipses and Circles

To draw an ellipse, use drawOval( ). To fill an ellipse, use fillOval( ). These methods are shown here:

void drawOval(int top, int left, int width, int height)
void fillOval(int top, int left, int width, int height)

```
// Draw
Ellipses
import
java.awt.*
; import
java.apple
t.*; /*
<applet code="Ellipses" width=300
height=200> </applet>
*/
public class Ellipses extends Applet
{ public void paint(Graphics g) {
g.drawOval(10, 10, 50, 50);
g.fillOval(100, 10, 75, 50);
g.drawOval(190, 10, 90, 30);
g.fillOval(70, 90, 140, 100);
} }
```

Drawing Arcs

Arcs can be drawn with drawArc( ) and fillArc( ), shown here:

```
void drawArc(int top, int left, int width, int height, int startAngle,int sweepAngle)
void fillArc(int top, int left, int width, int height, int startAngle,int sweepAngle)
```

The arc is bounded by the rectangle whose upper-left corner is specified by top,left and whose width and height are specified by width and height. The arc is drawn from startAngle through the angular distance specified by sweepAngle. Angles are specified in degrees.

Zero degrees is on the horizontal, at the three o' clock position. The arc is drawn counterclockwise if sweepAngle is positive, and clockwise if sweepAngle is negative. Therefore, to draw an arc from twelve o' clock to six o' clock, the start angle would be 90 and the sweep angle 180.

Drawing Polygons

It is possible to draw arbitrarily shaped figures using drawPolygon( ) and fillPolygon( ), shown here:

```
void drawPolygon(int x[ ], int y[ ], int numPoints)
void fillPolygon(int x[ ], int y[ ], int numPoints)
```

The polygon' s endpoints are specified by the coordinate pairs contained within the x and y arrays. The number of points defined by x and y is specified by numPoints. There are alternative forms of these methods in which the polygon is specified by a Polygon object.

The following applet draws several arcs:

```
//Draw Arcs
import
java.awt.*;
import
java.applet.*;
/*

<applet code="Arcs"
width=300 height=200>
</applet>
*/
public class Arcs extends Applet {
public void paint(Graphics g) {
g.drawArc(10, 40, 70, 70, 0, 75);
g.fillArc(100, 40, 70, 70, 0, 75);
g.drawArc(10, 100, 70, 80, 0, 175);
g.fillArc(100, 100, 70, 90, 0, 270);
g.drawArc(200, 80, 80, 80, 0, 180);
}
}
```

The following applet draws an hourglass shape:

```
// Draw Polygon
import
java.awt.*;
import
java.applet.*;
/*
<applet code="HourGlass"
width=230 height=210>
</applet>
*/
public class HourGlass extends Applet {
public void paint(Graphics g) {
int xpoints[] = {30, 200, 30, 200, 30};
int ypoints[] = {30, 30, 200, 200, 30};
int num = 5; g.drawPolygon(xpoints,
ypoints, num);
}
}
```

SWINGS

- ✓ Swing is a set of classes that provides more powerful and flexible components than are possible with the AWT. **Swing** is a GUI widget toolkit for **Java**. It is part of Oracle's **Java** Foundation Classes (JFC) that is *used to create window-based applications*. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.
- ✓ In addition to the familiar components, such as buttons, check boxes, and labels, Swing supplies several exciting additions, including tabbed panes, scroll panes, trees, and tables. Even familiar components such as buttons have more capabilities in Swing. For example, a button may have both an image and a text string associated with it. Also, the image can be changed as the state of the button changes.
- ✓ Unlike AWT components, Swing components are not implemented by platform specific code. Instead, they are written entirely in Java and, therefore, are platform-independent. The term *lightweight* is used to describe such elements.
- ✓ The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

Differences between AWT and Swing

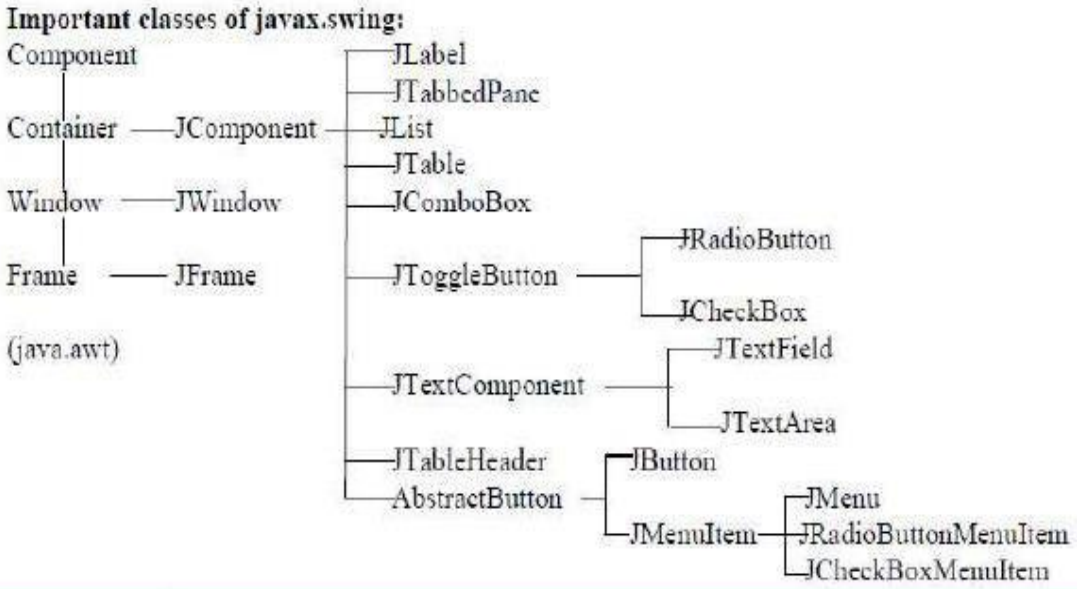
AWT	Swing
AWT components are called Heavyweight component.	Swings are called light weight component because swing components sits on the top of AWT components and do the work.
AWT components are platform dependent.	Swing components are made in purely java and they are platform independent.
AWT components require java.awt package.	Swing components require javax.swing package.
AWT is a thin layer of code on top of the OS.	Swing is much larger. Swing also has very much richer functionality.
AWT stands for Abstract windows toolkit.	Swing is also called as JFC's (Java Foundation classes).
This feature is not supported in AWT.	We can have different look and feel in Swing.
Using AWT, you have to implement a lot of things yourself.	Swing has them built in.

This feature is not available in AWT.	Swing has many advanced features like JTabel, Jtabbed pane which is not available in AWT. Also.Swing components are called "lightweight" because they do not require a native OS object to implement their functionality. JDialog and JFrame are heavyweight, because they do have a peer. So components like JButton, JTextArea, etc., are lightweight because they do not have an OS peer.

The Swing component classes are:

Class	Description
AbstractButton	Abstract superclass for Swing buttons.
ButtonGroup	Encapsulates a mutually exclusive set of buttons.
ImageIcon	Encapsulates an icon.
JApplet	The Swing version of Applet.
JButton	The Swing push button class.
JCheckBox	The Swing check box class.
JComboBox	Encapsulates a combo box (combination of a drop-down list & text field).
JLabel	The Swing version of a label.
JRadioButton	The Swing version of a radio button.
JScrollPane	Encapsulates a scrollable window.
JTabbedPane	Encapsulates a tabbed window.
JTable	Encapsulates a table-based control.
JTextField	The Swing version of a text field.
JTree	Encapsulates a tree-based control.

Hierarchy for Swing components:



JApplet
Fundamental to Swing is the **JApplet** class, which extends **Applet**. Applets that use Swing must be subclasses of **JApplet**. **JApplet** is rich with functionality that is not foundin **Applet**.

The content pane can be obtained via the method shown here:
Container getContentPane( )

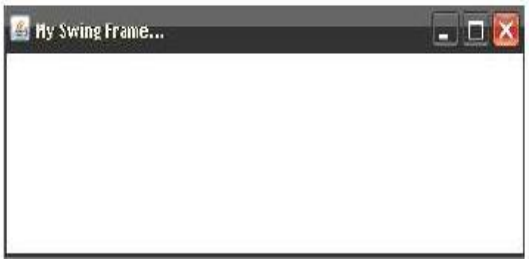
The **add( )** method of **Container** can be used to add a component to a content pane. Its form is shown here:
void add(*comp*)

Here, *comp* is the component to be added to the content pane.

JFrame
Create an object to JFrame: JFrame ob = new JFrame ("title"); (or)
Create a class as subclass to JFrame class: MyFrame extends JFrame
Create an object to that class : MyFrame ob = new MyFrame ();

Example: Write a program to create a frame by creating an object to JFrame class.

```
//A swing Frame
import javax.swing.*;
class MyFrame
{
public static void main (String agrs[])
{ JFrame jf = new JFrame ("My Swing Frame...");
jf.setSize (400,200);
jf.setVisible (true);
jf.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
}
}
```



Note: To close the frame, we can take the help of getDefaultCloseOperation () method of JFrame class, as shown here: **getDefaultCloseOperation (constant);**

where the constant can be any one of the following:

- JFrame.EXIT_ON_CLOSE: This closes the application upon clicking on close button.
- JFrame.DISPOSE_ON_CLOSE: This disposes the present frame which is visible on the screen. The JVM may also terminate.
- JFrame.DO_NOTHING_ON_CLOSE: This will not perform any operation upon clicking on close button.
- JFrame.HIDE_ON_CLOSE: This hides the frame upon clicking on close button.

Window Panes: In swings the components are attached to the window panes only. A window pane represents a free area of a window where some text or components can be displayed. For example, we can create a frame using JFrame class in javax.swing which contains a free area inside it, this free area is called 'window pane'. Four types of window panes are available in javax.swing package.

Glass Pane: This is the first pane and is very close to the monitors screen. Any components to be displayed in the foreground are attached to this glass pane. To reach this glass pane, we use getGlassPane () method of JFrame class.

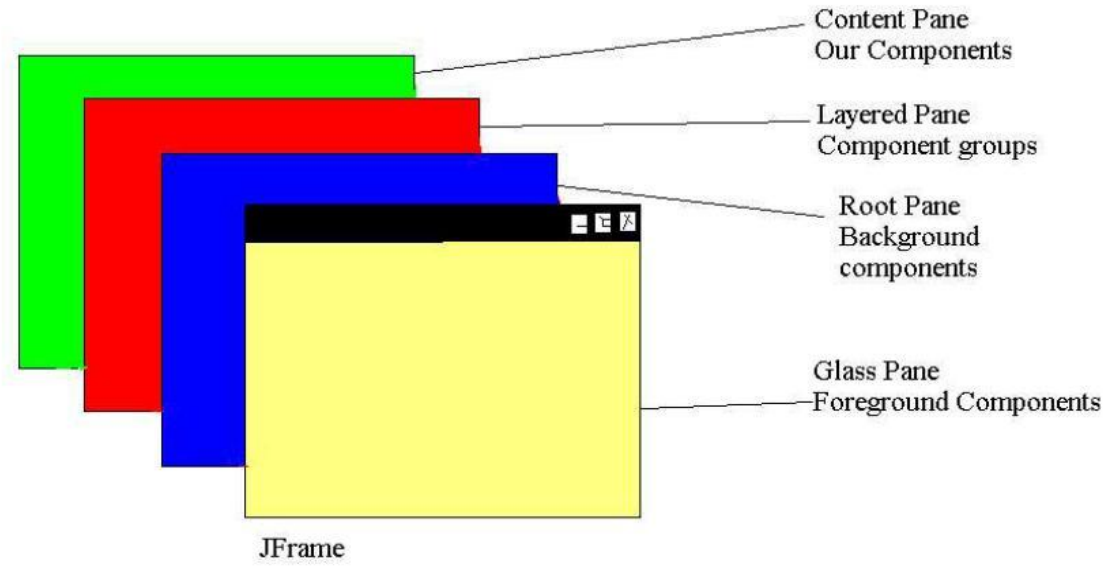
Root Pane: This pane is below the glass pane. Any components to be displayed in the background are displayed in this pane. Root pane and glass pane are used in animations also.
For example, suppose we want to display a flying aeroplane in the sky. The aeroplane can be displayed as a .gif or .jpg file in the glass pane where as the blue sky can be displayed in the root pane in the background. To reach this root pane, we use getRootPane () method of JFrame class.

Layered Pane: This pane lies below the root pane. When we want to take several components as a group, we attach them in the layered pane. We can reach this pane by calling getLayeredPane () method of JFrame class.

Content Pane: This is the bottom most pane of all. Individual components are attached to this pane. To reach this pane, we can call getContentPane () method of JFrame class.

Displaying Text in the Frame:

paintComponent (Graphics g) method of JPanel class is used to paint the portion of a component in swing. We should override this method in our class. In the following example, we are writing our class MyPanel as a subclass to JPanel and override the painComponent () method.



Write a program to display text in the frame

```
import javax.swing.*;
import java.awt.*;
class MyPanel extends JPanel
{ public void paintComponent (Graphics g)
{ super.paintComponent (g); //call JPanel's method
setBackground (Color.red);
g.setColor (Color.white);
g.setFont (new Font("Courier New",Font.BOLD,30));
```

```
g.drawString ("Hello Readers!", 50, 100);
}
}
class FrameDemo extends JFrame
{ FrameDemo ()
{
Container c = getContentPane ();
MyPanel mp = new MyPanel ();
c.add (mp);
setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
}
public static void main(String args[])
{ FrameDemo ob = new FrameDemo ();
ob.setSize (600, 200);
ob.setVisible (true);
}
}
```

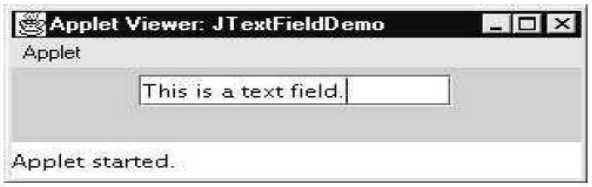
TEXT FIELDS

The Swing text field is encapsulated by the JTextComponent class, which extends JComponent. It provides functionality that is common to Swing text components. One of its subclasses is JTextField, which allows you to edit one line of text. Some of its constructors are shown here:

JTextField( )
JTextField(int cols)
JTextField(String s, int cols)
JTextField(String s)

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTextFieldDemo" width=300 height=50>
```

```
</applet>
*/
public class JTextFieldDemo extends JApplet  {
  JTextField jtf;
public void init()
{
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
   //  Add text field to content
      pane jtf = new
      JTextField(15);
      contentPane.add(jtf);
}
}
```



BUTTONS

Swing buttons provide features that are not found in the Button class defined by the AWT. For example, you can associate an icon with a Swing button. Swing buttons are subclasses of the AbstractButton class, which extends JComponent. AbstractButton contains many methods that allow you to control the behavior of buttons, check boxes, and radio buttons.

The JButton Class

The JButton class provides the functionality of a push button. JButton allows an icon, a string, or both to be associated with the push button. Some of its constructors are shown here:

- To create a JButton with text: JButton b = new JButton ("OK");
- To create a JButton with image: JButton b = new JButton (ImageIcon ii);
- To create a JButton with text & image: JButton b = new JButton ("OK", ImageIcon ii);

It is possible to create components in swing with images on it. The image is specified by ImageIcon class object.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JButtonDemo" width=250 height=300>
</applet>
*/
public class JButtonDemo extends JApplet implements ActionListener {
JTextField jtf;
public void init() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Add buttons to content pane
```

```
ImageIcon france = new ImageIcon("france.gif");
JButton jb = new JButton(france);
jb.setActionCommand("France");
jb.addActionListener(this); contentPane.add(jb);

ImageIcon germany = new ImageIcon("germany.gif");
jb = new JButton(germany);
jb.setActionCommand("Germany");
jb.addActionListener(this);
contentPane.add(jb);
ImageIcon italy = new ImageIcon("italy.gif");
jb = new JButton(italy);
jb.setActionCommand("Italy");
jb.addActionListener(this);
contentPane.add(jb);
ImageIcon japan = new ImageIcon("japan.gif");
jb = new JButton(japan);
jb.setActionCommand("Japan");
jb.addActionListener(this);
contentPane.add(jb);
      // Add text field to
            content pane jtf = new
            JTextField(15);
            contentPane.add(jtf);
}
public void actionPerformed(ActionEvent ae)
{ jtf.setText(ae.getActionCommand());
}
}
```

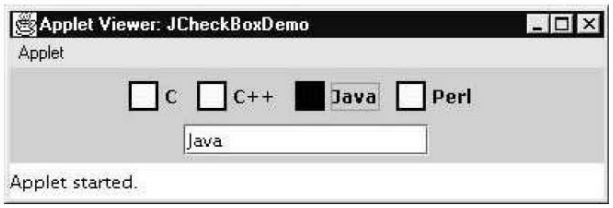
CHECK BOXES

The JCheckBox class, which provides the functionality of a check box, is a concrete implementation of AbstractButton. Its immediate superclass is JToggleButton, which provides support for two-state buttons. Some of its constructors are shown here:

```
JCheckBox(Icon i)
JCheckBox(Icon i, boolean state)
JCheckBox(String s)
JCheckBox(String s, boolean state)
JCheckBox(String s, Icon i)
JCheckBox(String s, Icon i, boolean state)
```

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JCheckBoxDemo" width=400 height=50>
</applet>
*/
```

```
public class JCheckBoxDemo extends JApplet implements ItemListener {
JTextField jtf;
public void init()
 {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Create icons
ImageIcon normal = new ImageIcon("normal.gif");
ImageIcon rollover = new ImageIcon("rollover.gif");
ImageIcon selected = new ImageIcon("selected.gif");
// Add check boxes to the content pane
JCheckBox cb = new JCheckBox("C", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this); contentPane.add(cb);

cb = new JCheckBox("C++",
normal); cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
cb = new JCheckBox("Java", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
cb = new JCheckBox("Perl", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
// Add text field to the content pane
jtf = new JTextField(15);
contentPane.add(jtf);
}
public void itemStateChanged(ItemEvent ie) {
JCheckBox cb = (JCheckBox)ie.getItem();
jtf.setText(cb.getText());
}
}
```

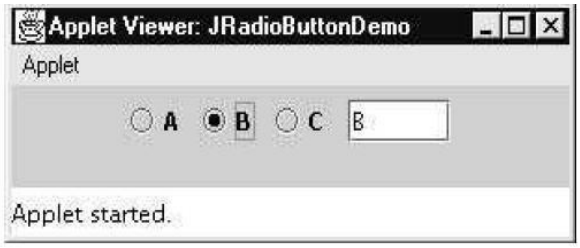


RADIO BUTTONS

Radio buttons are supported by the JRadioButton class, which is a concrete implementation of AbstractButton. Its immediate superclass is JToggleButton, which provides support for two-state buttons. Some of its constructors are shown here:

JRadioButton(Icon i)
JRadioButton(Icon i, boolean state)
JRadioButton(String s)
JRadioButton(String s, boolean state)
JRadioButton(String s, Icon i)
JRadioButton(String s, Icon i, boolean state)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JRadioButtonDemo" width=300 height=50>
</applet>
*/
public class JRadioButtonDemo extends JApplet implements ActionListener {
JTextField tf;
public void init() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
1. Add radio buttons to content pane
JRadioButton b1 = new JRadioButton("A");
b1.addActionListener(this);
contentPane.add(b1);
JRadioButton b2 = new JRadioButton("B");
b2.addActionListener(this);
contentPane.add(b2);
JRadioButton b3 = new JRadioButton("C");
b3.addActionListener(this);
contentPane.add(b3);
2. Define a button group
ButtonGroup bg = new ButtonGroup();
bg.add(b1);
bg.add(b2);
bg.add(b3);
// Create a text field and add it
// to the content pane
tf = new JTextField(5);
contentPane.add(tf);
}
public void actionPerformed(ActionEvent ae) {
tf.setText(ae.getActionCommand());
}
}
```



Limitations of AWT:

The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface. One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents or peers. This means that the look and feel of a component is defined by the platform, not by java. Because the AWT components use native code resources, they are referred to as heavy weight.

The use of native peers led to several problems.

First, because of variations between operating systems, a component might look, or even act, differently on different platforms. This variability threatened java's philosophy: write once, run anywhere.

Second, the look and feel of each component was fixed and could not be changed. Third, the use of heavyweight components caused some frustrating restrictions. Due to these limitations Swing came and was integrated to java. Swing is built on the AWT. Two key Swing features are: Swing components are light weight, Swing supports a pluggable look and feel.