

# **UNIT NO 1**

## **JAVA FUNDAMENTALS**

### **Introduction & Java Evolution (Java History):-**

Java is a general-purpose, object-oriented programming language developed by Sun Microsystems of USA in 1991. Originally called Oak by James Gosling, one of the inventors of the language, Java was designed for the development of software for consumer electronic devices like TVs, VCRS, toasters and such other electronic machines. The goal is to make the language simple, portable and highly reliable. The Java team discovered that the existing languages like C and C++ had limitations in terms of both reliability and portability. However, they modelled their new language Java on C and C++ but removed a number of features of C and C++ and thus made Java a really simple, reliable, portable, and powerful language.

### **Java Milestones:-**

<b>Year</b>	<b>Development</b>
1990	Sun Microsystems decided to develop special software that could be used to manipulate consumer electronic devices. A team of Sun Microsystems programmers headed by James Gosling was formed to undertake this task.
1991	The team announced a new language named "Oak".
1992	The team demonstrated the application of their new language to control a list of home appliances using a hand-held device with a tiny touch-sensitive screen.
1993	The World Wide Web (WWW) appeared on the Internet and transformed the text-based Internet into a graphical-rich environment. The Green Project team came up with the idea of developing Web applets (tiny programs) using the new language that could run on all types of computers connected to Internet.
1994	The team developed a Web browser called "HotJava" to locate and run applet programs on Internet.
1995	Oak was renamed "Java".
1996	Java established itself not only as a leader for Internet programming but also as a general-purpose, object-oriented programming language. Sun releases Java Development Kit 1.0.
1997	Sun releases Java Development Kit 1.1 (JDK 1.1).

1998	Sun releases the Java 2 with version 1.2 of the Software Development Kit (SDK 1.2). 1999 Sun releases Java 2 Platform, Standard Edition (J2SE) and Enterprise Edition (J2EE).
2000	J2SE with SDK 1.3 was released.
2002	J2SE with SDK 1.4 was released.
2004	J2SE with JDK 5.0 (instead of JDK 1.5) was released. This is known as J2SE 5.0.

The most striking feature of the language is that it is a platform-neutral language. Programs developed in Java can be executed anywhere on any system.

## **Java Features:-**

The java language is not only reliable, portable and distributed but also simple, compact and interactive.

- 1) Compiled and Interpreted
- 2) Platform-Independent and Portable
- 3) Object-Oriented
- 4) Robust and Secure
- 5) Distributed
- 6) Simple, Small and Familiar
- 7) Multithreaded and Interactive
- 8) High Performance Dynamic and Extensible

### **1) Compiled and Interpreted:-**

Usually a computer language is either compiled or interpreted. Java combines both these approaches first; Java compiler translates source code into what is known as byte code instructions. Byte codes are not machine instructions and therefore, in the second stage, Java interpreter converts the byte code to machine code. We can thus say that Java is both a compiled and an interpreted language.

### **2) Platform-Independent and Portable:-**

Java is a platform independent language because Java programs can be easily moved from one computer system to another, anywhere and anytime. This is the reason why Java has become a popular language for programming on internet. We can download a Java applet from a remote computer onto our local system via Internet and execute it locally.

**3) Object-Oriented:-**

Java is a true object-oriented language. Almost everything in Java is an object. All program code and data reside within objects and classes. Java comes with a large set of classes, arranged in packages that we can use in our programs by inheritance.

**4) Robust and Secure:-**

Java is a robust language. It provides many safeguards to ensure reliable code. It has strict compile time and run time checking for data types. Java also consist the concept of exception handling which captures the runtime errors and eliminates any risk of crashing the system. Java systems not only verify all memory access but also ensure that no viruses are communicated with an applet.

**5) Distributed:-**

Java is designed as a distributed language for creating applications on networks. It has the ability to share both data and programs. Java applications can open and access remote objects on Internet. This enables multiple programmers at multiple remote locations to collaborate and work together on a single project.

**6) Simple, Small and Familiar:-**

Java is a small and simple language. Many features of C and C++ that are either redundant or sources of unreliable code are not part of Java. For example, Java does not use pointers, preprocessor header files, goto statement and many others. It also eliminates operator overloading and multiple inheritance.

**7) Multithreaded:-**

Multithreaded means handling multiple tasks simultaneously. Java supports multithreaded programs. For example, we can listen to an audio clip while scrolling a page and at the same time download an applet from a computer.

**8) High Performance:-**

Java performance is impressive for an interpreted language, mainly due to the use of intermediate byte code. According to Sun, Java speed is comparable to the native C/C++.

## Difference between Java and C++:- (Java vs C++)

There are many differences and similarities between the C++ programming language and Java.

Comparison Index	C++	Java
<b>Platform-independent</b>	C++ is platform-dependent.	Java is platform-independent.
<b>Mainly used for</b>	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in Windows-based, web-based and mobile applications.
<b>Goto</b>	C++ supports the goto statement.	Java doesn't support the goto statement.
<b>Multiple inheritance</b>	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by using interfaces in java.
<b>Operator Overloading</b>	C++ supports operator overloading.	Java doesn't support operator overloading.
<b>Pointers</b>	C++ supports pointers.	Java does not use pointer.
<b>Compiler and Interpreter</b>	C++ uses compiler only.	Java uses both compiler and interpreter.
<b>Call by Value and Call by reference</b>	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.
<b>Structure and Union</b>	C++ supports structures and unions.	Java doesn't support structures and unions.
<b>Thread Support</b>	C++ doesn't have built-in support for threads.	Java has built-in thread support.

## Simple Java Program:-

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

        System.out.println("Java is better than C++");
    }
}
```

In above program we are using following

- 1) **Class Declaration:** - The first line **class Welcome** declares a class, which is an object-oriented construct. Everything in java must be placed inside a class. Class is a keyword and declares that a new class. Welcome is a Java identifier that specifies the name of the class to be defined.
- 2) **Opening Brace:** - Every class definition in Java begins with an opening brace { and ends with a matching closing brace }. This is similar to C++ class construct.
- 3) **The Main Line:** - The third line **public static void main (String args[])** defines a method named main. This is similar to the main() function in C/C++. A Java application can have any number of classes but only one of them must include a main method to start the execution.

This line contains a number of keywords, public, static and void.

**public:** The keyword public is an access specifier that declares the main method as public and therefore making it accessible to all other classes.

**static:** Next appears the keyword static, which declares this method as one that belongs to the entire class and not a part of any objects of the class.

**void:** The type modifier void states that the main method does not return any value.

**String args[]:** declares a parameter named args, which contains an array of objects of the class type String.

- 4) **The Output Line:** - The only executable statement in the program is

**System.out.println("Java is better than C++");**

This is similar to the printf() statement of C or cout << construct of C++. The println method is a member of the out object, which is a static data member of System class.

This line prints the string "Java is better than C++"

## Internal Path Setting in Java:

The path is required to be set for using tools such as javac, java, etc. If you are saving the Java source file inside the JDK/bin directory, the path is not required to be set because all the tools will be available in the current directory. However, if you have your Java file outside the JDK/bin folder, it is necessary to set the path of JDK.

There are two ways to set the path in Java:

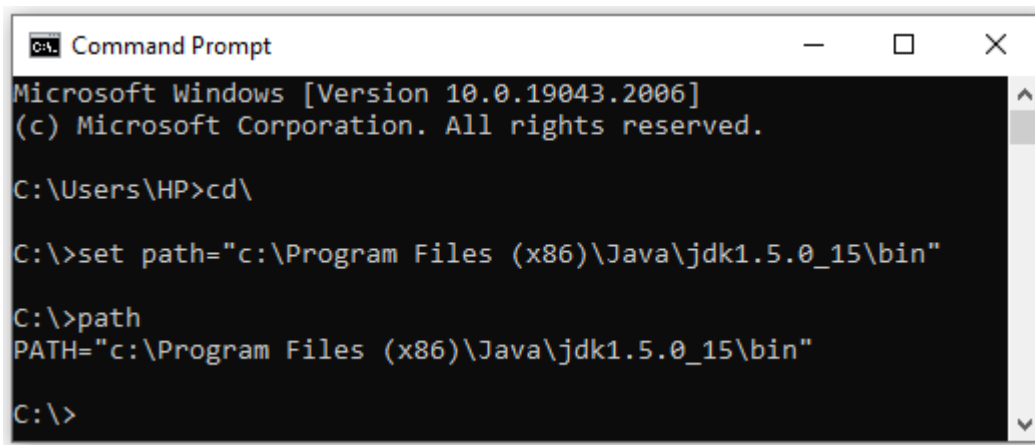
1. Temporary
2. Permanent

### 1) How to set the Temporary Path of JDK in Windows:

To set the temporary path of JDK, you need to follow the following steps:

- Open the command prompt
- Copy the path of the JDK/bin directory
- Write in command prompt: set path=copied\_path

**For Example:** set path=C:\Program Files (x86)\Java\jdk1.5.0\_15\bin



```
C:\>Microsoft Windows [Version 10.0.19043.2006]
(c) Microsoft Corporation. All rights reserved.

C:\Users\HP>cd\

C:\>set path="c:\Program Files (x86)\Java\jdk1.5.0_15\bin"

C:\>path
PATH="c:\Program Files (x86)\Java\jdk1.5.0_15\bin"

C:\>
```

### 2) How to set Permanent Path of JDK in Windows:

For setting the permanent path of JDK, you need to follow these steps:

Go to MyComputer properties -> advanced tab -> environment variables -> new tab of user variable -> write path in variable name -> write path of bin folder in variable value -> ok -> ok -> ok

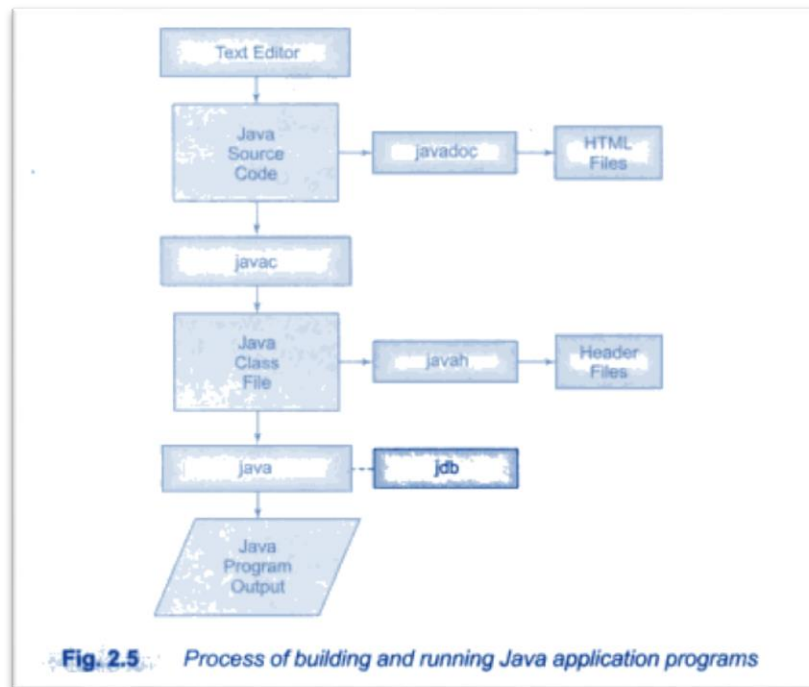
- 1) Go to MyComputer properties (Right Click on MyComputer).
- 2) Click on the advanced tab.
- 3) Click on environment variables.

- 4) Click on the new tab of user variables.
- 5) Write the path in the variable name.
- 6) Copy the path of bin folder.
- 7) Paste path of bin folder in the variable value.
- 8) Click on Ok button.

### **JDK (Java Development Kit):-**

The Java Development Kit comes with a collection of tools that are used for developing and running Java programs. They include: Following Table lists these tools and their descriptions.

<b>Tool</b>	<b>Description</b>
Appletviewer	Enables us to run Java applets (without actually using a Java-compatible browser).
Java	Java interpreter, which runs applets and applications by reading and interpreting bytecode files
Javac	The Java compiler, which translates Java sourcecode to bytecode files that the interpreter can understand.
Javadoc	Creates HTML-format documentation from Java source code files
Javah	Produces header files for use with native methods.
javap	Java disassembler, which enables us to convert bytecode files into a program description.
Jdb	Java debugger, which helps us to find errors in our programs



### JRE (Java Run-time Environment):-

Java Run-time Environment (JRE) is the part of the Java Development Kit (JDK). It is a freely available software distribution which has Java Class Library, specific tools, and a stand-alone JVM. It is the most common environment available on devices to run java programs. The source Java code gets compiled and converted to Java bytecode. If you wish to run this bytecode on any platform, you require JRE. The JRE loads classes, verify access to memory, and retrieves the system resources. JRE acts as a layer on the top of the operating system.

#### JRE consists of the following components:

- Technologies which get used for deployment such as Java Web Start.
- Toolkits for user interface like Java 2D (2 dimensional graphics e.g. 2D games)
- Integration libraries like **Java Database Connectivity (JDBC)** and **Java Naming and Directory Interface (JNDI)**.
- Libraries such as Lang and util.

#### Working of JRE:

Java Development Kit (JDK) and Java Runtime Environment (JRE) both interact with each other that enables Java-based applications to run on any platform (operating system). The JRE runtime architecture consists of the following elements:



1. ClassLoader
  2. ByteCode verifier
  3. Interpreter
- **ClassLoader:** Java ClassLoader dynamically loads all the classes necessary to run a Java program. Because classes are only loaded into memory whenever they are needed, the JRE uses ClassLoader will automate this process when needed.
  - **Bytecode Verifier:** The bytecode checker ensures the format of Java code before passing it to the interpreter. If the code violates system integrity or access rights, the class is considered corrupt and will not load.
  - **Interpreter:** After loading the byte code successfully, the Java interpreter creates an object of the Java virtual machine that allows the Java program to run natively on the underlying machine.

**In this way, the program runs in JRE**

### **JVM (Java Virtual Machine):-**

All language compilers translate source code into machine code for a specific computer. Java compiler also does the same thing. Then, how does Java achieve architecture neutrality? The answer is that the Java compiler produces an intermedia code known as bytecode for a machine that does not exist. This machine is called the Java Virtual Machine and it exists only inside the computer memory. Below Figure shows that the process of compiling a Java program into bytecode which is also referred to as virtual machine code.



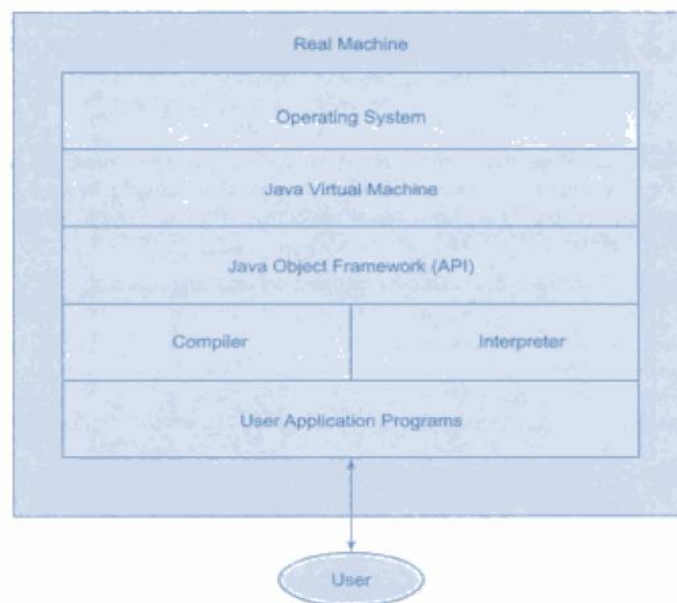
**Fig. 3.6** *Process of compilation*

The virtual machine code is not machine specific. The machine specific code is generated by the Java interpreter by acting as an intermediary between the virtual machine and the real machine. Remember that the interpreter is different for different machines.



**Fig. 3.7** *Process of converting bytecode into machine code*

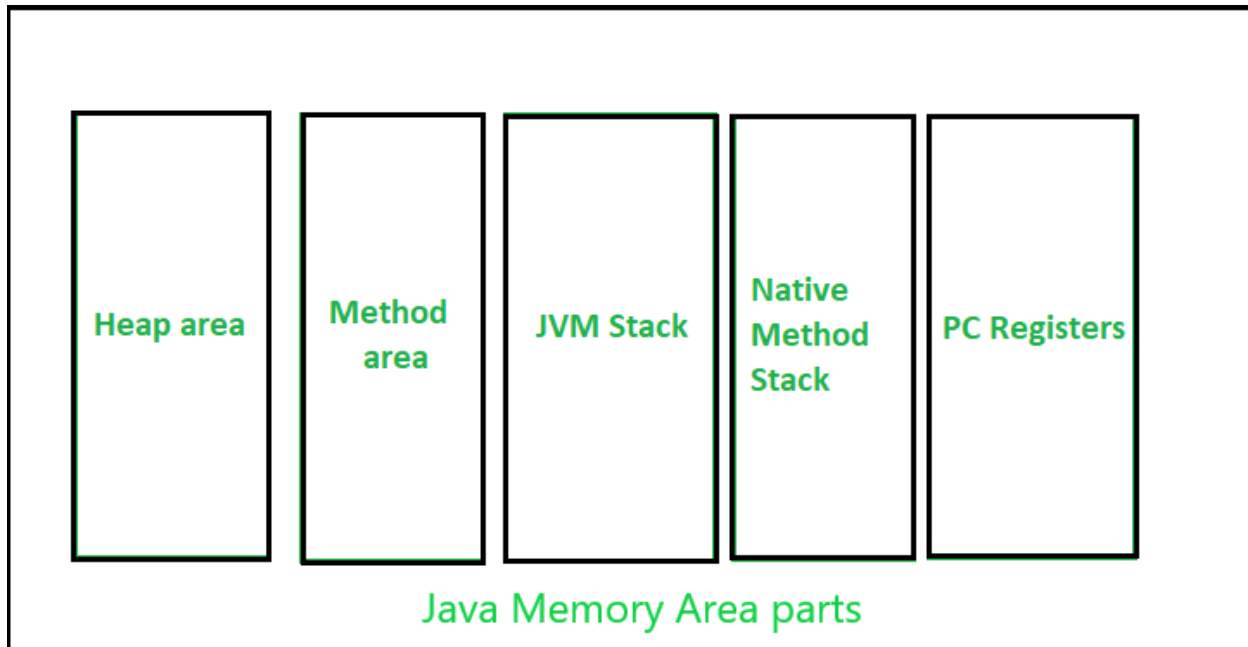
The Java object framework (Java API) acts as the intermediary between the user programs and the virtual machine which in turn acts as the intermediary between the operating system and the Java object framework.



**Fig. 3.8** *Layers of interactions for Java programs*

## JVM Memory Management:

In java, JVM (Java Virtual Machine) is responsible for managing memory allocation and deallocation to ensure efficient memory usage. The memory is divided into different regions.



### 1) Heap:

- It is a runtime data area and stores the actual object in a memory.
- This memory is allocated for all class instances and array. Heap can be of fixed or dynamic size depending upon the system's configuration.
- JVM provides the user control to initialize or vary the size of heap as per the requirement. When a new keyword is used, object is assigned a space in heap memory.
- There exists one and only one heap for a running JVM process.

*E.g. `Scanner sc = new Scanner(System.in);`*

- The above statement creates the object of Scanner class which gets allocated to heap whereas the reference 'sc' gets pushed to the stack.
- *Note: Garbage collection in heap area is mandatory.*

### 2) Method Area:

- It is a logical part of the heap area and is created on virtual machine startup.
- This memory is allocated for class structures, method data and constructor field data, and also for interfaces or special method used in class.
- Can be of a fixed size or expanded as required by the computation. Needs not to be contiguous.
- ***Note:** Though method area is logically a part of heap, it may or may not be garbage collected even if garbage collection is compulsory in heap area.*

**3) JVM Stacks:**

- Each thread in java application has its own stack, which is used to store method invocations (calling or requesting the execution of a method) and local variables.
- The stack memory is automatically managed by the JVM and is generally smaller in size compared to heap.
- When a method is called, a new frame is created on the stack to store the methods variables and data. The frame is removed when the method completes its execution.

**4) Native method Stacks:**

Also called as C stacks, native method stacks are not written in Java language. This memory is allocated for each thread when it's created. And it can be of fixed or dynamic nature.

**5) Program counter (PC) registers:**

Each JVM thread which carries out the task of a specific method has a program counter register associated with it. The non native method has a PC which stores the address of the available JVM instruction whereas in a native method, the value of program counter is undefined. PC register is capable of storing the return address or a native pointer on some specific platform.

**Java Unicode:**

Computer systems internally store data in binary representation. A character is stored using a combination of 0's and 1's. The process is called encoding. A character encoding scheme is important because it helps to represent the same information on multiple types of devices.

**Types of Encoding:**

Following are the different types of encoding used before the Unicode system.

1. ASCII (American Standard Code for Information Interchange): used for the United States
2. ISO 8859-1 used for the Western European Languages
3. KOI-8 used for Russian
4. GB18030 and BIG-5 used for Chinese and so on.
5. Base64 used for binary to text encoding

## What is Unicode System?

- Unicode system is an international character encoding technique that can represent most of the languages around the world.
- Unicode System is established by Unicode Consortium.
- Hexadecimal values are used to represent Unicode characters.
- There are multiple Unicode Transformation Formats:
  - UTF-8: It represents 8-bits (1 byte) long character encoding.
  - UTF-16: It represents 16-bits (2 bytes) long character encoding
  - UTF-32: It represents 32-bits (4 bytes) long character encoding.
- To access a Unicode character the format starts with an escape sequence `\u` followed by 4 digits hexadecimal value.
- A Unicode character has a range of possible values starting from `\u0000` to `\uFFFF`.
- Some of the Unicode characters are
  - `\u00A9` represent the copyright symbol - ©
  - `\u0394` represent the capital Greek letter delta - Δ
  - `\u0022` represent a double quote - "

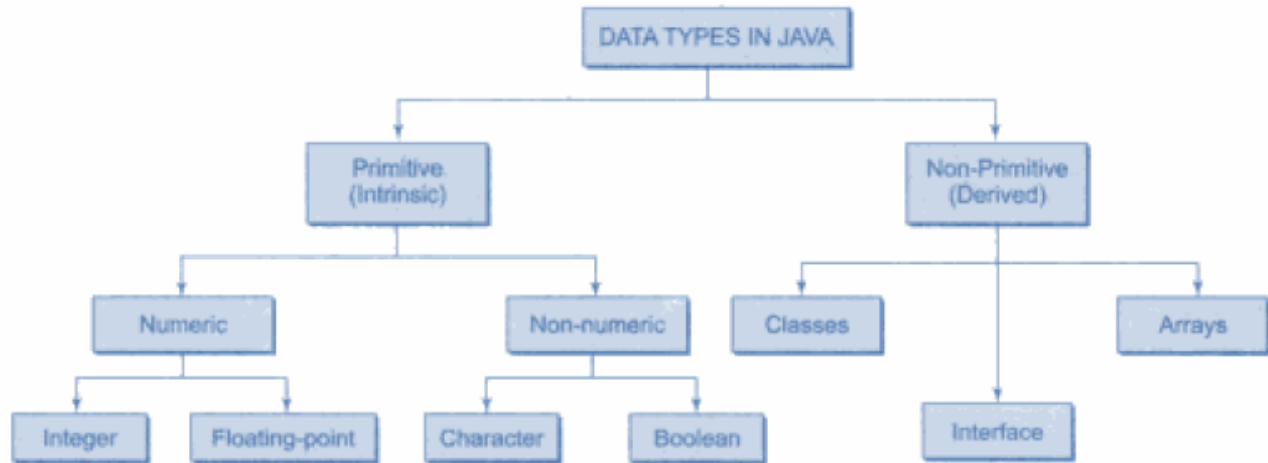
## Java Keywords:-

Java language has 50 keywords. Keywords have specific meaning in Java. We cannot use them as names for variables, classes, methods and so on. All keywords are to be written in lower-case letters.

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

## Data Types:-

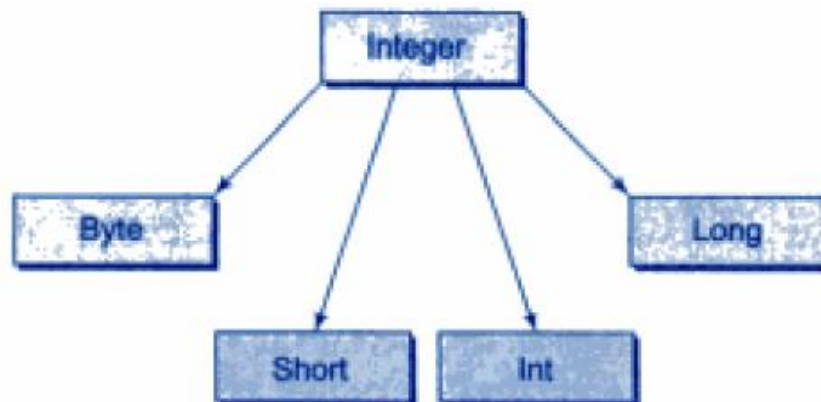
Every variable in Java has a data type. Data types specify the size and type of values that can be stored. Java language is rich in its data types. Data types in Java under various categories are



**Fig. 4.2** Data types in Java

## Integer Types:-

Integer types can hold whole numbers such as 123, -96, and 5639. Java supports four types of integers. They are byte, short, int, and long.



**Fig. 4.3** Integer data types

Type	Size	Minimum Value	Maximum Value
byte	1 byte	-128	127
short	2 byte	-32,768	32,767

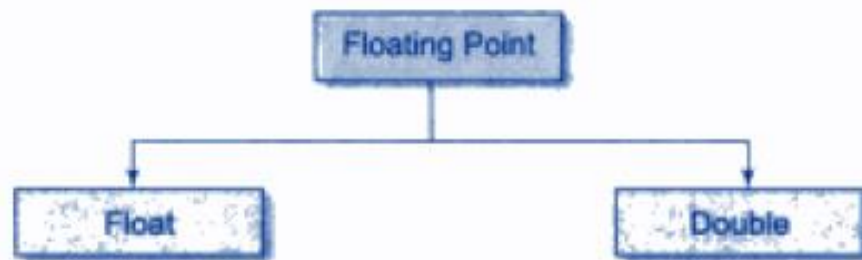
int	4 byte	-2, 147, 483, 648	2, 147, 483, 647
long	8 byte	-9, 223, 372, 036, 854, 775, 808	9, 223, 372, 036, 854, 775, 807

We can make integers long by appending the letter L or l at the end of the number.

**Example: 123L or 123l**

### Floating Point Types:-

Integer types can hold only whole numbers and therefore we use another type known as floating point type to hold numbers containing fractional parts such as 27.59 and -1.375 (known as floating point constants). There are two kinds of floating point storage in Java.



**Fig. 4.4** Floating point data types

The float type values are single-precision numbers while the double types represent double precision numbers. In single precision mode, we must append f or F to the numbers. **Example: 1.231f or 7.56923e5F.** In double precision mode, we must append d or D to the numbers. **Example: 1.231d or 7.56923e5D.**

Type	Size	Minimum Value	Maximum Value
float	4 byte	3.4e-038	3.4e+038
double	8 byte	1.7e-308	1.7e+308

Double-precision types are used when we need greater precision in storage of floating point numbers. All mathematical functions, such as sin, cos and sqrt return double type values.

### Character Type:-

In order to store character constants in memory, Java provides a character data type called char. The char type assumes a size of 2 bytes but, basically, it can hold only a single character. **Example: 'S' 'a'**

### Boolean Type:-

Boolean type is used when we want to test a particular condition during the execution of the program. There are only two values that a boolean type can take: true or false. Boolean type is denoted by the keyword boolean and uses only one bit of storage.

### Operators:-

Java supports a rich set of operators. An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. Java operators can be classified into a number of related categories as below:

- 1) Arithmetic operators
- 2) Relational operators
- 3) Logical operators
- 4) Assignment operators
- 5) Increment and decrement operators
- 6) Conditional operators
- 7) Bitwise operators
- 8) Special operators

#### 1) Arithmetic Operators:-

Arithmetic operators are used to construct mathematical expressions. Java provides all the basic arithmetic operators.

Operator	Meaning	Example
+	Addition	a+b
-	Subtraction	a-b
*	Multiplication	a*c
/	Division	a/b
%	Modulo Division (Remainder)	a%b



## 2) Relational Operators:-

We often compare two quantities, and depending on their relation, take certain decisions. For example, we may compare the age of two persons, or the price of two items, and so on. These comparisons can be done with the help of relational operators.

Operator	Meaning	Example
<	is less than	a<b
<=	is less than or equal to	a<=b
>	is greater then	a>b
>=	is greater than or equal to	a>=b
==	is equal to	a==b
!=	is not equal to	a!=b

## 3) Logical Operators:-

The logical operators which combines two or more relational expressions is called as logical expression. Java has three logical operators.

Operator	Meaning	Example
&&	Logical AND	a>10 && a<20
	Logical OR	a>10    a<20
!	Logical NOT	a!=10

## 4) Assignment Operators:-

Assignment operators are used to assign the value of an expression to a variable. In addition, Java has a set of shorthand assignment operators which are used in the form

**Syntax: v op= exp;      or      v=v op(exp)**

where v is a variable, exp is an expression and op is a Java binary operator. The operator op is known as the shorthand assignment operator.

Statement with simple assignment operator	Statement with shorthand operator
a=a+1	a+=1

$a=a-1$	$a-=1$
$a=a*(n+1)$	$a*=n+1$
$a=a/(n+1)$	$a/=n+1$
$a=a\%b$	$a\%=1$

### 5) Increment and Decrement Operators:-

Java has two very useful operators. These are the increment and decrement operators: ++ and --. The operator ++ adds 1 to the operand while -- subtracts 1. Both operators are used in the following form:

Syntax: ++m or m++ is equivalent to m=m+1 or m+=1

### 6) Conditional Operator:-

The character pair ? : is a ternary operator available in Java. This operator is used to construct conditional expressions of the form.

**Syntax: expl ? exp2 : exp3**

where expl, exp2, and exp3 are expressions.

### 7) Bitwise Operators:-

Java has a distinction of supporting special operators known as bitwise operators for manipulation of data at values of bit level. These operators are used for testing the bits, or shifting them to the right or left. Bitwise operators may not be applied to float or double.

Operator	Meaning
<b>&amp;</b>	bitwise AND
<b> </b>	bitwise OR
<b>^</b>	bitwise exclusive OR
<b>~</b>	one's complement
<b>&lt;&lt;</b>	shift left
<b>&gt;&gt;</b>	shift right
<b>&gt;&gt;&gt;</b>	shift right with zero fill

### 8) Special Operators:-

Java supports some special operators of interest such as instanceof operator and member selection operator (.).

1. **Instanceof Operator:-** The instanceof is an object reference operator and returns true if the object on the left-hand side is an instance of the class given on the right-hand side. This operator allows us to determine whether the object belongs to a particular class or not.

**Example: person instanceof student**

is true if the object person belongs to the class student; otherwise it is false.

2. **Dot Operator:-** The dot operator (.) is used to access the instance variables and methods of class objects.

**Examples: person1.age;** // Reference to the variable age

**person1.salary();** // Reference to the method salary()

It is also used to access classes and sub-packages from a package.

## Control Statements in Java:

A Java program is a set of statements, which are normally executed in sequential order. A programming language uses control statements to control the flow of execution of a program based on certain conditions.

Java provides three types of control flow statements.

- 1) Decision Making Statements

- if statements
- switch statement

- 2) Loop Statements

- do while loop
- while loop
- for loop
- for-each loop

- 3) Jump Statements

- break statement
- continue statement

### 1) Decision Making and Branching:-

When a program breaks the sequential flow and jumps to another part of the code, it is called branching. When the branching is based on a particular condition, it is known as conditional branching. Java language supports the following statements known as control or decision making statements.

- i) if statement
- ii) switch statement

### i) If Statement:-

It allows the computer to evaluate the expression first and then, depending on whether the value of the expression is 'true' or "false", it transfers the control to a particular statement.

The if statement may be implemented in different forms

- 1. Simple if statement
- 2. if..else statement
- 3. Nested if..else statement
- 4. else if ladder

#### 1. Simple If Statement:-

The general form of a simple if statement is

##### Syntax:

```
if(test expression)
{
    statement-block;
}
statement-x;
```

If the test expression is true, the statement-block will be executed; otherwise the statement-block will be skipped and the execution will jump to the statement-x.

##### Flowchart of simple if control:

##### Example:

```
class SimpleIf
{
    public static void main(String args[])
    {
        int a=5;
        int b=3;
        if(a>b)
        {
            System.out.println("A is greater than B");
        }
    }
}
```

```
    }  
    System.out.println("End of program");  
}  
}
```

2. **The If...Else Statement:-** If the test expression is true, then the true-block statement(s) are executed; otherwise, the false-block statement(s) are executed.

**Syntax:**

```
if(test expression)  
{  
    True-block statement(s)  
}  
else  
{  
    False-block statement(s)  
}  
statement-x
```

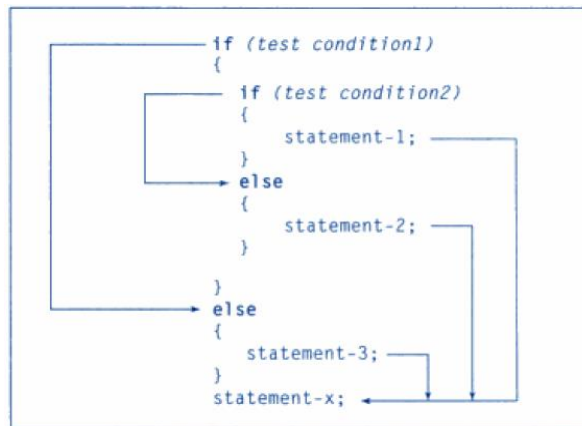
**Example:**

```
class IfElse  
{  
    public static void main(String args[])  
    {  
        int a=5;  
        int b=3;  
        if(a>b)  
        {  
            System.out.println("A is greater than B");  
        }  
        else  
        {  
            System.out.println("B is greater than A");  
        }  
        System.out.println("End of program");  
    }  
}
```

3. **Nesting of If...Else Statements:-**

When a series of decisions are involved, we may have to use more than one if...else statement in nested form. If the condition-1 is false, the statement-3 will be executed; otherwise it continues to perform the second test. If the condition-2 true, the

statement-1 will be evaluated; otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x.

**Syntax:****Example:**

class NestingIfElse

```
{
    public static void main(String args[])
    {
        int a=2,b=5,c=7;
        if(a>b)
        {
            if(a>c)
            {
                System.out.println("A is greater than B and C");
            }
            else
            {
                System.out.println("C is greater than A and B");
            }
        }
        else
        {
            if(c>b)
            {
                System.out.println("C is greater than A and B");
            }
            else
            {
                System.out.println("B is greater than A and C");
            }
        }
    }
}
```

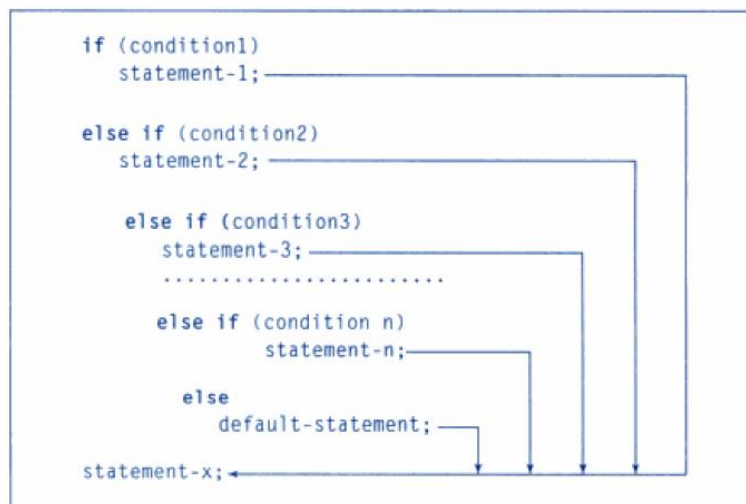
```

    }
}
System.out.println("End of program");
}
}

```

- 4. The Else If Ladder:** - Else If Ladder is a multipath decision is a chain of ifs this construct is known as the else if ladder. The conditions are evaluated from the top to downwards. As soon as the true condition is found, the statement associated with it is executed and the control is transferred to the statement-x. When all the n conditions become false, then the final else containing the default-statement will be executed.

**Syntax:**



**Example:**

```

class ElseIfLadder
{
    public static void main(String args[])
    {
        int a=8,b=5,c=7;
        if(a>b && a>c)
        {
            System.out.println("A is greater than B and C");
        }
        else if(b>a && b>c)
        {
            System.out.println("B is greater than A and C");
        }
        else if(c>a && c>b)

```

```
        {  
            System.out.println("C is greater than A and B");  
        }  
        System.out.println("End of program");  
    }  
}
```

## 5. The Switch Statement:-

Java has a built-in multi way decision statement known as switch. The switch statement tests the value of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case executed.

### Syntax:

```
switch (expression)  
{  
    case value-1:  
        block-1  
        break;  
    case value-2:  
        block-2  
        break;  
    .....  
    .....  
    default:  
        default-block  
        break;  
}  
statement-x;
```

### Example:

```
class SwitchCase  
{  
    public static void main(String args[])  
    {  
        int month=6;  
        switch(month)  
        {  
            case 1:  
                System.out.println("Jan");  
                break;  
            case 2:  
                System.out.println("Feb");  
                break;  
        }  
    }  
}
```



```
case 3:
    System.out.println("Mar");
    break;
case 4:
    System.out.println("Apr");
    break;
case 5:
    System.out.println("May");
    break;
case 6:
    System.out.println("Jun");
    break;
case 7:
    System.out.println("Jul");
    break;
case 8:
    System.out.println("Aug");
    break;
case 9:
    System.out.println("Sep");
    break;
case 10:
    System.out.println("Oct");
    break;
case 11:
    System.out.println("Nov");
    break;
case 12:
    System.out.println("Dec");
    break;
default:
    System.out.println("Not a valid Month!");
}
}
```

## 2) Looping:-

The process of repeatedly executing a block of statements is known as looping. The statements in the block may be executed any number of times, from zero to infinite number. If a loop continues forever, it is called an infinite loop.

The Java language provides for three constructs for performing loop operations. They are:

1. while construct
2. do while construct
3. for construct

#### 1. The While Statement:-

The simplest of all the looping structures in Java is the while statement. The basic format of the while statement is

```
Initialization:
While (test condition)
{
    Body of the loop
}
```

The while is an entry-controlled loop statement. The test condition is evaluated and if the condition is true, then the body of the loop is executed. This process of repeated execution of the body continues until the test condition finally becomes false and the control is transferred out of the loop. The body of the loop may have one or more statements.

#### Example:

```
class WhileTest
{
    public static void main(String args[])
    {
        int i;
        i=1; //Initialization
        while(i<=10) // Condition
        {
            System.out.println("Number is : "+i);
            i++; //Increment
        }
    }
}
```

#### 2. The do Statement:-

The do statement program proceeds to evaluate the body of the loop first. At the end of the loop, the test condition in the while statement is evaluated. If the condition is true, the program continues to evaluate the body of the loop once again. When the condition

becomes false, the loop will be terminated. The do....while construct provides an exit-controlled loop and therefore the body of the loop is always executed at least once.

```
Initialization:
do
{
    Body of the loop
}
while (test condition):
```

**Example:**

```
class DoWhileTest
{
    public static void main(String args[])
    {
        int i;
        i=1; //Initialization
        do
        {
            System.out.println("Number is : "+i);
            i++; //Increment
        }
        while(i<=10); //Condition
    }
}
```

**3. The for Statement:-**

The for loop is another entry-controlled loop that provides a more concise loop control structure. The general form of the for loop is

```
for (initialization ; test condition ; increment)
{
    Body of the loop
}
```

- Initialization of the control variables is done first, using assignment statements such as i=1.

- The test condition is a relational expression, such as `i<10` that determines when the loop will exit. If the condition is true, the body of the loop is executed; otherwise the loop is terminated.
- Now, the control variable is incremented or decremented using an assignment statement such as `i=i+1` or `i++`.

**Example:**

```
class ForTest
{
    public static void main(String args[])
    {
        int i;
        for(i=1;i<=10;i++)
        {
            System.out.println("Number is : "+i);
        }
    }
}
```

## 2) Jump Statements:-

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

### 1. Java Break Statement:

The break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

**Example:**

```
class BreakExample
{
    public static void main(String args[])
    {
        for(int i = 0; i<= 10; i++)
        {
            System.out.println(i);
        }
    }
}
```

```
        if(i==6)
        {
            break;
        }
    }
}
```

## 2. Continue Statement:

Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

### Example:

```
class ContinueExample
{
    public static void main(String args[])
    {
        for(int i = 0; i<= 10; i++)
        {
            System.out.println(i);
            if(i==6)
            {
                continue;
            }
        }
    }
}
```

## Classes, Objects and Methods:-

**Definition of Class:** A class is a collection of objects of similar types.

Or

A class is a collection of data members and member functions

A class is a user-defined data type that helps to define its properties. Once the class type has been defined, we can create variables. In Java, these variables are termed as instances of classes. The basic form of a class definition is:

```
class Classname [extends superclassname]  
{  
    [ fields declaration; ]  
    [ method declaration; ]  
}
```

The keyword `extends` indicates that the properties of the `superclassname` class are extended to the `classname` class. This concept is known as inheritance. Fields and methods are declared inside the body.

### Example:

```
class Student  
{  
}
```

## Fields Declaration:-

We can declare the instance variables exactly the same way as we declare local variables.

### Example:

```
class Rectangle  
{  
    int length;    // Fields Declaration  
    int width;  
}
```

The class `Rectangle` contains two integer type instance variables. Remember these variables are only declared and therefore no storage space has been created in the memory. Instance variables are also known as member variables.

## Methods Declaration:-

Methods are declared inside the body of the class but immediately after the declaration of instance variables. The general form of a method declaration is

```
type methodname (parameter-list)
{
    method-body;
}
```

Method declarations have four basic parts:

1. The name of the method (methodname)
2. The type of the value the method returns (type)
3. A list of parameters (parameter-list)
4. The body of the method

### Example:

```
class Rectangle
{
    int length;    // Field Declaration
    int width;
    void getData(int x, int y) // Method Declaration
    {
        length=x;
        width=y;
    }
    int rectArea() // Declaration of another method
    {
        int area;
        area=length*width;
        return(area);
    }
}
```

## Creating Objects:-

Creating an object is also referred to as instantiating an object. Objects in Java are created using the new operator. Example of creating an object of type Rectangle.

**Rectangle rect1; // declare the object**

**rect1=new Rectangle(); // instantiate the object                      OR**

Both statements can be combined into one: **Rectangle rect1=new Rectangle();**

## Accessing Class Members:-

We cannot access the instance variables and the methods directly. To do this, we must use the concerned object and the dot(.) operator as shown below:

```
objectname.variablename =value;
objectname.methodname (parameter-list);
```

### Example:

```
Rectangle rect1=new Rectangle();
rect1.length=15;
rect1.width=10;
rect1.getData(10,20);
```

### Program: Application of classes and objects

```
class Rectangle
{
    int length;    // Field Declaration
    int width;
    void getData(int x, int y) // Method Declaration
    {
        length=x;
        width=y;
    }
    int rectArea() // Declaration of another method
    {
        int area;
        area=length*width;
        return(area);
    }
}
class RectArea
{
    public static void main(String args[])
    {
        int area1,area2;
        Rectangle rect1=new Rectangle();
        rect1.length=10;
        rect1.width=20;
        area1=rect1.length*rect1.width;
        System.out.println("Area =" +area1);
        rect1.getData(10,20);
        area2=rect1.rectArea();
        System.out.println("Area =" +area2);
    }
}
```



## Constructors:-

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

### Rules for creating java constructor

There are basically two rules defined for the constructor.

1. Constructors have the same name as the class itself.
2. They do not specify a return type, not even void.

### Types of java constructors

There are two types of constructors:

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

#### 1) **Default Constructor:**

A constructor that has no parameter is known as default constructor. Default constructor refers to a constructor that is automatically created by compiler in the absence of explicit constructors.

You can also call a constructor without parameters as default constructor because all of its class instance variables are set to default values.

#### Syntax of default constructor:

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

### Example of default constructor:-

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
class Bike
{
    Bike()
    {
        System.out.println("Bike is created");
    }
    public static void main(String args[])
    {
        Bike b=new Bike();
    }
}
```

**Output: Bike is created**

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

## **2) Java parameterized constructor:-**

A constructor that have parameters is known as parameterized constructor. Why use parameterized constructor? Parameterized constructor is used to provide different values to the distinct objects.

### **Example of parameterized constructor**

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
class Student
{
    int id;
    String name;
    Student(int i, String n)
    {
        id = i;
        name = n;
    }
    void display()
    {
        System.out.println(id+" "+name);
    }
    public static void main(String args[])
    {
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan");
        s1.display();
        s2.display();
    }
}
```

Output:

```
111 Karan
222 Aryan
```

**Difference between constructor and method in java:-**

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

**Access Modifiers in Java:**

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**. The access modifiers in Java specify the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc.

**Understanding Java Access Modifiers:**

Access Modifier	within class	within package	outside package by subclass only	outside package
<b>Private</b>	Y	N	N	N
<b>Default</b>	Y	Y	N	N
<b>Protected</b>	Y	Y	Y	N
<b>Public</b>	Y	Y	Y	Y

1) **Private:** The private access modifier is accessible only within the class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

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

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

2) **Default:** If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

```
//save by A.java
package pack;
```

```
class A
{
    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();//Compile Time Error
        obj.msg();//Compile Time Error
    }
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

- 3) Protected:** The protected access modifier is accessible within package and outside the package but through inheritance only. The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

```
//save by A.java
package pack;
public class A
{
    protected void msg()
    {
        System.out.println("Hello");
    }
}
//save by B.java
package mypack;
import pack.*;
class B extends A
{
    public static void main(String args[])
    {
```

```
        B obj = new B();
        obj.msg();
    }
}
```

- 4) **Public:** The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.

**//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();
    }
}
```

## Static Keyword:

The **static keyword** in Java is mainly used for memory management. The static keyword in Java is used to share the same variable or method of a given class. The users can apply static keywords with variables, methods, blocks, and nested classes. The *static* keyword is a non-access modifier in Java.

1. Blocks
2. Variables
3. Methods
4. Classes

**Note:** To create a static member (block, variable, method, nested class), use keyword **static**.

- 1) **Static variable:** If you declare any variable as static, it is known as a static variable. When a variable is declared as static, then a single copy of the variable is created and shared among all objects at the class level. Static variables are, essentially, global variables. All instances of the class share the same static variable.

**Example:**

```
class StaticVariable
{
    // static variable
    static int a = 10;
    // non-static variable
    int b = 5;
}
class StaticVariableDemo
{
    public static void main(String[] args)
    {
        // access the static variable
        System.out.println("Static Variable= " + StaticVariable.a);
        StaticVariable obj = new StaticVariable();
        // access the non-static variable
        System.out.println("Non Static Variable=" + obj.b);
    }
}
```

- 2) **Static Block:** **Static block is used to initialize the static data member. It is executed before the main method at the time of class loading.**

**Example:**

```
class StaticBlockDemo
{
    static
    {
        System.out.println("Static block is invoked");
    }
    public static void main(String args[])
    {
        System.out.println("Hello main");
    }
}
```

**3) Static methods: When a method is declared with the static keyword, it is known as the static method. The most common example of a static method is the main() method.**

**Methods declared as static can have the following restrictions:**

- They can directly call other static methods only.
- They can access static data directly.
- A static method can access static data member and can change the value of it.

**Example:**

```
class Student
{
    int rollno;
    String name;
    static String college = "SGM";
    //static method to change the value of static variable
    static void change()
    {
        college = "BDC";
    }
    //constructor to initialize the variable
    Student(int r, String n)
    {
        rollno = r;
        name = n;
    }
    //method to display values
    void display()
    {
        System.out.println(rollno+" "+name+" "+college);
    }
}
public class StaticMethodDemo
{
    public static void main(String args[])
    {
        //Student.change();//calling change method
        //creating objects
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan");
        //calling display method
        s1.display();
    }
}
```



```
        s2.display();
    }
}
```

- 4) **Static Classes:** A class can be made **static** only if it is a nested class. We cannot declare a top-level class with a static modifier but can declare nested classes as static. Such types of classes are called Nested static classes. Nested static class doesn't need a reference of Outer class. In this case, a static class cannot access non-static members of the Outer class.

**Example:**

```
class StaticClassDemo
{
    private static String str = "Balasaheb Desai College, Patan";
    // Static class
    static class MyNestedClass
    {
        // non-static method
        public void display()
        {
            System.out.println(str);
        }
    }
    public static void main(String args[])
    {
        StaticClassDemo.MyNestedClass obj= new
        StaticClassDemo.MyNestedClass();
        obj.display();
    }
}
```

**Final Keyword:**

The final keyword is a non-access modifier used for classes, attributes and methods, which makes them non-changeable. The java final keyword can be used in many context. Final can be used in

- 1) variable
- 2) method
- 3) class

- 1) **Java final variable:** When a variable is declared with the final keyword, its value can't be modified, essentially, a constant. This also means that you must initialize a final variable.

**Example:**

```
class FinalVariable
{
    final int ProductId=90;//final variable
    void display()
    {
        ProductId=100; // cannot change final variable
        System.out.println("Product ID="+ProductId);
    }
    public static void main(String args[])
    {
        FinalVariable obj=new FinalVariable();
        obj.display();
    }
}
```

Output: Compile Time Error

- 2) **Java final Method:** Before you learn about final methods and final classes, make sure you know about the Java Inheritance. In Java, the final method cannot be overridden by the child class.

**Example:**

```
class Bike
{
    final void display()
    {
        System.out.println("running");
    }
}
class Honda extends Bike
{
    void display() //cannot override method
    {
        System.out.println("running safely with 100kmph");
    }
    public static void main(String args[])
    {
        Honda honda= new Honda();
        honda.display();
    }
}
```

Output: Compile Time Error

**Note:** final method is inherited but you cannot override it.

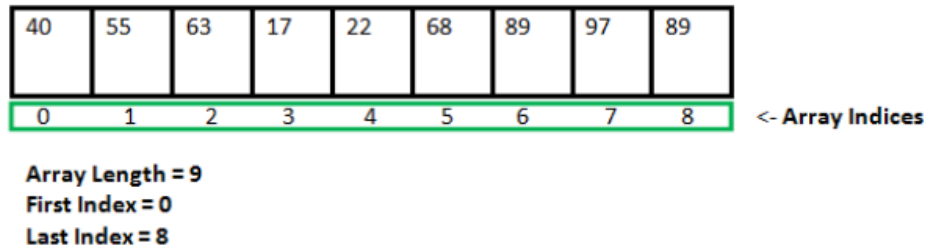
- 3) Final classes: When a class is declared with final keyword, it is called a final class. A final class cannot be extended (inherited).

**Example:**

```
// create a final class
final class FinalClass
{
    public void display()
    {
        System.out.println("This is a final method.");
    }
}
// try to extend the final class
class MainClass extends FinalClass
{
    public void display()
    {
        System.out.println("The final method is overridden.");
    }
    public static void main(String[] args)
    {
        MainClass obj = new MainClass();
        obj.display();
    }
}
```

## Arrays:

An array is a collection of similar type of elements. Java array is an object which contains elements of a similar data type. Additionally, the elements of an array are stored in a contiguous memory location. Since arrays are objects in Java, we can find their length using the object property length.



**Creating an Array:** To declare an array, define the variable type with **square brackets**.

**Syntax:** `type var-name[];` OR `type[] var-name;`

**Example:** `int arr[];` OR `int[] arr;`

**Instantiating an Array:** When an array is declared, only a reference of an array is created. To create or give memory to the array, you create an array like

**Syntax:** `var-name = new type [size];`

**Example:**

```
int arr[]; //declaring array  
arr = new int[20]; // allocating memory to array  
  
OR  
  
int[] arr = new int[20]; // combining both statements in one
```

**Accessing Elements in Array:** You can access an array element by referring to the index number. The index begins with 0 and ends at (total array size)-1. All the elements of array can be accessed using Java for Loop.

**Example:**

```
class ArrayDemo  
{  
    public static void main(String[] args)  
    {  
        // declares an Array of integers.  
        int[] arr;  
        // allocating memory for 5 integers.  
        arr = new int[5];  
    }  
}
```

```
// initialize the elements of the array
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
arr[3] = 40;
arr[4] = 50;
// accessing the elements of the specified array
for (int i = 0; i < arr.length; i++)
{
    System.out.println("Element at index " + i + " : " + arr[i]);
}
}
```

**Array of Objects:** Array of Objects stores objects that mean objects are stored as elements of an array.

**Syntax:** `Class_Name[ ] objectArrayReference;`

**Example:** `Student[] obj = new Student[5];` //student is a user-defined class

**Example:**

```
class Student
{
    public int rollno;
    public String name;
    Student(int rollno, String name)
    {
        this.rollno = rollno;
        this.name = name;
    }
}
// Elements of the array are objects of a class Student.
class StudentArrayDemo
{
    public static void main(String[] args)
    {
        // declares an Array of integers.
        Student[] arr;
        // allocating memory for 5 objects of type Student.
        arr = new Student[5];
        // initialize the elements of the array
        arr[0] = new Student(1, "aman");
        arr[1] = new Student(2, "vaibhav");
    }
}
```

```
arr[2] = new Student(3, "shikar");
arr[3] = new Student(4, "dharmesh");
arr[4] = new Student(5, "mohit");
// accessing the elements of the specified array
for (int i = 0; i < arr.length; i++)
{
    System.out.println("Element at " + i + " : " + arr[i].rollno + " "+
arr[i].name);
}
}
```

### Types of Array in java:

**There are two types of array.**

- 1) Single Dimensional Array
- 2) Multidimensional Array

#### 1) Single Dimensional Array in Java:

An array with one dimension is called one-dimensional array or single dimensional array in java. It is a list of variables (called elements or components) containing values that all have the same type.

**Syntax:** dataType[] arr; or dataType arr[];

##### Example:

```
class SingleDimensionalArray
{
    public static void main(String args[])
    {
        int a[]={ 10,20,30,40,50}; //declaration and instantiation
        for(int i=0;i<a.length;i++)
        {
            System.out.println(a[i]);
        }
    }
}
```

#### 2) Multidimensional Array in Java:

In such case, data is stored in row and column based index (also known as matrix form).

**Syntax:** dataType[][] arrayRefVar; OR dataType arrayRefVar[][];

##### Example:

```
int[][] myArray={{ 10, 20, 30},{ 11, 21, 31},{ 12, 22, 32}}; //3 row and 3 column
```

	Col0	Col1	Col2
Row0	10	20	30
Row1	11	21	31
Row2	12	22	32

```
class MultidimensionalArray
{
    public static void main(String args[])
    {
        //declaring and initializing 2D array
        int arr[][]={{ 10,20,30},{ 11,21,31},{ 12,22,32 }};
        for(int i=0;i<3;i++)
        {
            for(int j=0;j<3;j++)
            {
                System.out.print(arr[i][j]+" ");
            }
            System.out.println();
        }
    }
}
```

## String Manipulation in Java:

String manipulation is a sequence of characters. They are widely used in Java. For example, "hello" is a string containing a sequence of characters 'h', 'e', 'l', 'l', and 'o'. We use **double quotes** to represent a string in Java. For example,

// create a string

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

### Example of a simple Java String

```
public class SimpleString
{
    public static void main(String args[])
    {
        //creating a string by java string literal
        String str = "Modern India ";
        char ch[]={'B','C','A'};
```

```

        //converting the char array ch[] to string str2
        String str2 = new String(ch);
        //creating another java string 'str3' by using new keyword
        String str3 = new String("String Example");
        //Displaying all the three strings
        System.out.println(str);
        System.out.println(str2);
        System.out.println(str3);
    }
}

```

**Output:**

Modern India

BCA

String Example

**String Length**

The Java String class contains a length() method that **returns the total number of characters a given String contains**. This value includes all blanks, spaces, and other special characters.

```

public class LengthExample
{
    public static void main(String args[])
    {
        String s1="javatpoint";
        String s2="python";
        System.out.println("string length is: "+s1.length());//10 is the length of javatpoint
string
        System.out.println("string length is: "+s2.length());//6 is the length of python
string
    }
}
string length is: 10
string length is: 6

```

**Methods of Java String:**

**Java String** class provides a lot of methods to perform operations on strings

Methods	Description
<u>contains()</u>	checks whether the string contains a substring class ContainsMethod { public static void main(String[] args)



	<pre>         {             String str1 = "Java String contains()";             // check if str1 contains "Java"             boolean result = str1.contains("Java");             System.out.println(result);         }     }     // Output: true </pre>
<u>substring()</u>	<p>returns the substring of the string</p> <p>class SubstringMethod</p> <pre> {     public static void main(String[] args)     {         String str1 = "java is fun";         // extract substring from index 0 to 3         System.out.println(str1.substring(0, 4));     } } // Output: java </pre>
<u>replace()</u>	<p>replaces the specified old character with the specified new character</p> <p>class ReplaceMethod</p> <pre> {     public static void main(String[] args)     {         String str1 = "bat ball";         // replace b with c         System.out.println(str1.replace('b', 'c'));     } } // Output: cat call </pre>
<u>charAt()</u>	<p>returns the character present in the specified location</p> <p>class CharAtMethod</p> <pre> {     public static void main(String[] args)     {         String str1 = "Java Programming";         // returns character at index 2         System.out.println(str1.charAt(2));     } } // Output: v </pre>

<u>indexOf()</u>	<p>returns the position of the specified character in the string.</p> <pre> class IndexOfMethod {     public static void main(String[] args)     {         String str1 = "Java is fun";         int result;         // getting index of character 's'         result = str1.indexOf('s');         System.out.println(result);     } } // Output: 6 </pre>
<u>trim()</u>	<p>removes any leading and trailing whitespaces</p> <pre> class TrimMethod {     public static void main(String[] args)     {         String str1 = "  Learn Java Programming  ";         System.out.println(str1.trim());     } } // Output: Learn Java Programming </pre>
<u>toLowerCase()</u>	<p>converts the string to lowercase</p> <pre> class LowerCaseMethod {     public static void main(String[] args)     {         String str1 = "JAVA PROGRAMMING";         // convert to lower case letters         System.out.println(str1.toLowerCase());     } } // Output: java programming </pre>
<u>toUpperCase()</u>	<p>converts the string to uppercase</p> <pre> class UpperCaseMethod {     public static void main(String[] args)     {         String str1 = "Learn Java";         String str2 = "Java123"; </pre>

	<pre>// convert to upper case letters System.out.println(str1.toUpperCase()); // "LEARN JAVA" System.out.println(str2.toUpperCase()); // "JAVA123"     } }</pre>
--	--

## **UNIT NO 2**

### **INHERITANCE, POLYMORPHISM AND ENCAPSULATION**

#### **Introduction:**

Reusability is yet another aspect of OOP paradigm. It is always nice if we could reuse something that already exists rather than creating the same all over again, Java supports this concept. Java classes can be reused in several ways. This is basically done by creating new classes, reusing the properties of existing ones.

#### **Defining a Subclass:-**

A subclass is defined as follows:

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

The keyword `extends` signifies that the properties of the superclassname are extended subclassname. The subclass will now contain its own variables and methods as well as those of the superclass.

#### **Inheritance: Extending a Class**

**Definition:** The mechanism of deriving a new class from an old one is called inheritance. The old class is known as the base class or super class or parent class and the new one is called the subclass or derived class or child class. The inheritance allows subclasses to inherit all the variables and methods of their parent classes.

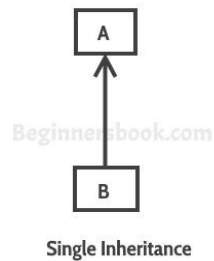
#### **Types of Inheritance:**

- 1) Single inheritance (only one super class)
- 2) Multiple inheritance (several super classes)
- 3) Hierarchical inheritance (one super class, many subclasses)
- 4) Multilevel inheritance (Derived from a derived class)

Java does not directly implement multiple inheritance. However, this concept is implemented using an interfaces.

## 1) Single Inheritance:

When a class inherits another class, it is known as a *single inheritance*. Single inheritance refers to a child and parent class relationship where a class extends another class.



### Single Inheritance Example:

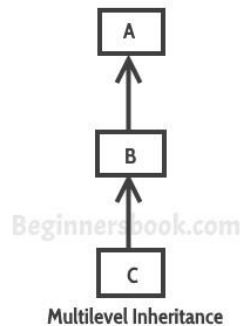
```
class A
{
    void methodA()
    {
        System.out.println("Method of Class A");
    }
}

class B extends A
{
    void methodB()
    {
        System.out.println("Method of Class B");
    }
}

class SingleInheritance
{
    public static void main(String args[])
    {
        B obj=new B();
        obj.methodA(); // Calling superclass method
        obj.methodB(); // Calling local methods
    }
}
```

## 2) Multilevel Inheritance:

When there is a chain of inheritance, it is known as multilevel inheritance. Multilevel inheritance: refers to a child and parent class relationship where a class extends the child class. For example class C extends class B and class B extends class A.



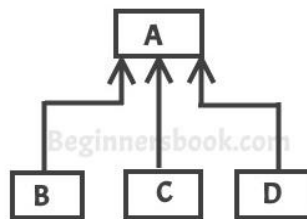
### Multilevel Inheritance Example:

```
class A
{
    void methodA()
    {
        System.out.println("Method of Class A");
    }
}
class B extends A
{
    void methodB()
    {
        System.out.println("Method of Class B");
    }
}
class C extends B
{
    void methodC()
    {
        System.out.println("Method of Class C");
    }
}
class MultilevelInheritance
{
    public static void main(String args[])
    {
        C obj=new C();
```

```
        obj.methodA();  
        obj.methodB();  
        obj.methodC();  
    }  
  
}
```

### 3) Hierarchical Inheritance:

When two or more classes inherits a single class, it is known as hierarchical inheritance. Hierarchical inheritance: refers to a child and parent class relationship where more than one classes extends the same class. For example, classes B, C & D extends the same class A.



Hierarchical Inheritance

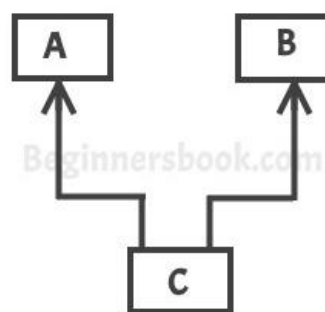
#### Hierarchical Inheritance Example:

```
class A  
{  
    void methodA()  
    {  
        System.out.println("Method of Class A");  
    }  
}  
class B extends A  
{  
    void methodB()  
    {  
        System.out.println("Method of Class B");  
    }  
}  
class C extends A  
{  
    void methodC()  
    {  
        System.out.println("Method of Class C");  
    }  
}
```

```
class D extends A
{
    void methodD()
    {
        System.out.println("Method of Class D");
    }
}
class HierarchicalInheritance
{
    public static void main(String args[])
    {
        B obj=new B();
        obj.methodA();
        obj.methodB();
        C obj1=new C();
        obj1.methodA();
        obj1.methodC();
        D obj2=new D();
        obj2.methodA();
        obj2.methodD();
    }
}
```

#### 4) Multiple Inheritance:

Multiple Inheritance refers to the concept of one class extending more than one classes, which means a child class has two parent classes. For example class C extends both classes A and B. Java doesn't support multiple inheritance.

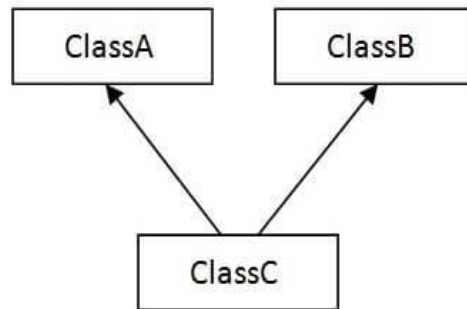


Multiple Inheritance

### Interfaces: Multiple Inheritance

Java does not support multiple inheritance. That is, classes in Java cannot have more than one superclass. For instance, a definition like





It is not permitted in Java. Java provides an alternate approach known as interfaces to support the concept of multiple inheritance. Although a Java class cannot be a subclass of more than one superclass, it can implement more than one interface.

### Defining Interface:

An interface is basically a kind of class. Like classes, interfaces contain methods and variables but with a major difference. The difference is that interfaces define only abstract methods and final fields.

The syntax for defining an interface is very similar to that for defining a class. The general form of

```
interface InterfaceName
{
    variables declaration;
    methods declaration;
}
```

**Here is an example of an interface definition that contains two variables and one method:**

```
interface Item
{
    static final int code=1001;
    static final String name ="Fan";
    void display ( );
}
```

## Extending Interfaces:

Like classes, interfaces can also be extended. That is, an interface can be subinterfaced from other interfaces. The new subinterface will inherit all the members of the superinterface in the manner similar to subclasses. This is achieved using the keyword `extends` as shown below:

```
interface name2 extends name1
{
    body of name2;
}
```

### Example:

```
interface ItemConstants
{
    int code=1001;
    string name="Fan";
}
interface Item extends ItemConstants
{
    void display();
}
```

## Implementing Interfaces:

Interfaces are used as "superclasses" whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. This is done as follows:

```
class classname implements interfacename
{
    body of classname;
}
```

## Program implementing interfaces:

```
interface Area
{
    final static float pi=3.14F;
```

```
        float compute (float x, float y);
    }
    class Rectangle implements Area
    {
        public float compute (float x, float y)
        {
            return (x*y);
        }
    }

    class Circle implements Area
    {
        public float compute (float x, float y)
        {
            return (pi*x*x);
        }
    }
    class InterfaceTest
    {
        public static void main(String args[])
        {
            Rectangle rect=new Rectangle();
            Circle cir=new Circle();
            Area area;
            area=rect; // area refers to rect object
            System.out.println("Area of Rectangle =" +area.compute(10,20));
            area=cir;
            System.out.println("Area of Circle =" +area.compute(10, 0));
        }
    }
```

## Super Keyword in Java:-

The **super** keyword in java is a reference variable that is used to refer to parent class objects. The keyword “super” came into the picture with the concept of Inheritance. The keyword super is used subject to the following conditions.

- super may only be used within a subclass constructor method
- The call to superclass constructor must appear as the first statement within the subclass constructor.

- The parameters in the super call must match the order and type of the instance variable declared in the superclass.

It is majorly used in the following contexts:

1. Use of super with variables
2. Use of super with methods
3. Use of super with constructors

**1) Use of super with variables:** This scenario occurs when a derived class and base class has the same data members.

**Example:**

// Java code to show use of super keyword with variables

// Base class vehicle

class Vehicle

{

int maxSpeed = 120;

}

// sub class Car extending vehicle

class Car extends Vehicle

{

int maxSpeed = 180;

void display()

{

// print maxSpeed of base class (vehicle)

System.out.println("Maximum Speed: "+ super.maxSpeed);

}

}

// Driver Program

class Test

{

public static void main(String[] args)

{

Car small = new Car();

small.display();

}

}

**Output:** Maximum Speed:120

- 2) **Use of super with variables:** The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
// Java program to show use of super with methods
// superclass Person
class Person
{
    void message()
    {
        System.out.println("This is person class");
    }
}
// Subclass Student
class Student extends Person
{
    void message()
    {
        System.out.println("This is student class");
    }
    // Note that display() is only in Student class
    void display()
    {
        message(); // will invoke or call current class message() method
        super.message(); // will invoke or call parent class message() method
    }
}
class StudentTest
{
    public static void main(String args[])
    {
        Student s = new Student();
        s.display();// calling display() of Student
    }
}
```

**Output:**

This is student class  
This is person class

- 3) **Use of super with constructors:** The super keyword can also be used to access the parent class constructor.

**Example:**

```
class Person
{
    Person()
    {
        System.out.println("Person class Constructor");
    }
}
// subclass Student extending the Person class
class Student extends Person
{
    Student()
    {
        super(); // invoke or call parent class constructor
        System.out.println("Student class Constructor");
    }
}
class StudentTestDemo
{
    public static void main(String[] args)
    {
        Student s = new Student();
    }
}
```

**Output:**

Person class Constructor  
Student class Constructor

## Polymorphism:

Polymorphism is another important OOP concept. Polymorphism means ability to take more than one form. The word “poly” means many and “morphs” means forms. So polymorphism means many forms. In other words, polymorphism allows you define interface and have multiple implementations.

There are two types of polymorphism

- 1) Compile time polymorphism – Static binding
- 2) Run-time polymorphism – dynamic binding

Method overloading is an example of Compile time polymorphism and method overriding is an example of Run-time polymorphism.

### 1) Methods Overloading:-

In Java, it is possible to create methods that have the same name, but different parameter lists and different definitions. This is called method overloading. Method overloading is used

when objects are required to perform similar tasks but using different input parameters. When we call a method in an object, Java matches up the method name first and then the number and type of parameters to decide which one of the definitions to execute. This process is known as polymorphism. To create an overloaded method, all we have to do is to provide several different method definitions in the class, all with the same name, but with different parameter lists. The difference may either be in the number or type of arguments. That is, each parameter list should be unique. Note that the method's return type does not play any role in this.

**Example of creating an overloaded method:**

```
class Calculator
{
    int add(int x,int y)
    {
        return(x+y);
    }
    int add(int x,int y,int z)
    {
        return(x+y+z);
    }
}
class MethodOverloading
{
    public static void main(String args[])
    {
        int a,b;
        Calculator obj=new Calculator();
        a=obj.add(10,20);
        System.out.println("Addition of Two Number : "+a);
        b=obj.add(10,20,30);
        System.out.println("Addition of Three Number : "+b);
    }
}
```

**2) Overriding Methods:-**

We have seen that a method defined in a super class is inherited by its subclass and is used by the objects created by the subclass. Method inheritance enables us to define and use methods repeatedly in subclasses without having to define the methods again in subclass.

However, there may be occasions when we want an object to respond to the same method but have different behavior when that method is called. That means, we should override the method defined in the superclass. This is possible by defining a method in the subclass that has the same name, same arguments and same return type as a method in the superclass. Then, when that method is called, the method defined in the subclass is invoked and executed instead of the one in the superclass. This is known as overriding.

**Example of method overriding:**

```
class Super
{
    int x;
    Super(int x)
    {
        this.x=x;
    }
    void display()
    {
        System.out.println("Super x = "+x);
    }
}
class Sub extends Super
{
    int y;
    Sub (int x, int y)
    {
        super(x);
        this.y=y;
    }
    void display()
    {
        System.out.println("Super x=" + x);
        System.out.println("Sub y =" +y);
    }
}
class MethodOverride
{
    public static void main(String args[])
    {
        Sub s1=new Sub(10,20);
        s1.display();
    }
}
```



```

    }
}

```

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

## this keyword:

- **this** is a keyword in Java. Which can be used inside method or constructor of class.
- It (**this**) works as a reference to current object whose method or constructor is being invoked.
- **this** keyword can be used to refer any member of current object from within an instance method or a constructor.
- The most common use of this keyword is to eliminate the confusion between class attributes and parameters with the same name.

Syntax of using this keyword

**this.varName**

varName is a name of an instance variable.

e.g.

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

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

    public static void main(String args[])
    {
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan");
        s1.display();
        s2.display();
    }
}
```

**Output 111 Karan  
222 Aryan**

## UNIT NO 3

### **Package, Multithreading and Exception handling**

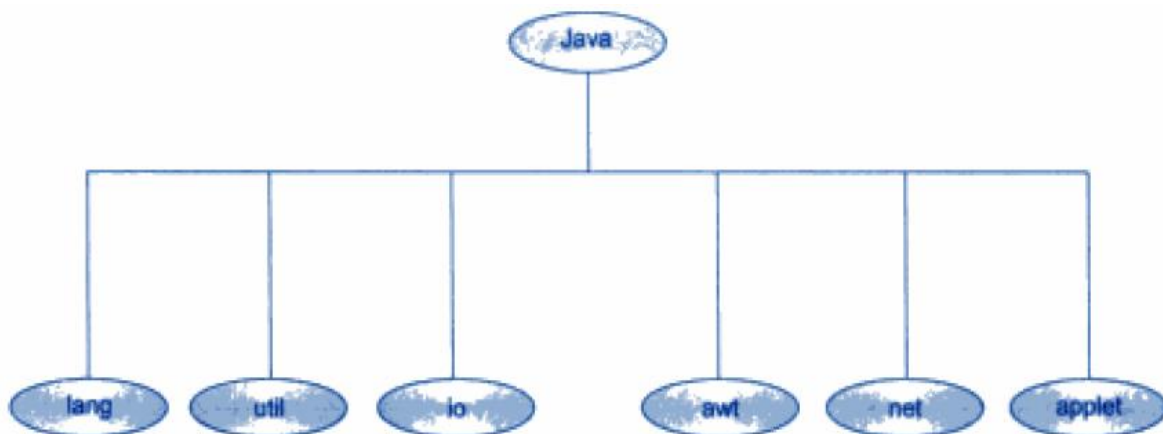
#### **Packages: Putting Classes Together:**

Packages are Java's way of grouping a variety of classes and/or interfaces together. The grouping is usually done according to functionality. In fact, packages act as "containers" for classes. By organizing our classes into packages we achieve the following benefits:

1. The classes contained in the packages of other programs can be easily reused.
2. In packages, classes can be unique compared with classes in other packages. That is, two classes in two different packages can have the same name. They may be referred by their fully qualified name, comprising the package name and the class name.
3. Packages provide a way to "hide" classes thus preventing other programs or packages from accessing classes that are meant for internal use only.
4. Packages also provide a way for separating "design" from "coding". First we can design classes and decide their relationships, and then we can implement the Java code needed for the methods.

#### **Java API Packages:**

Java API provides a large number of classes grouped into different packages according to functionality. Most of the time we use the packages available with the Java API.

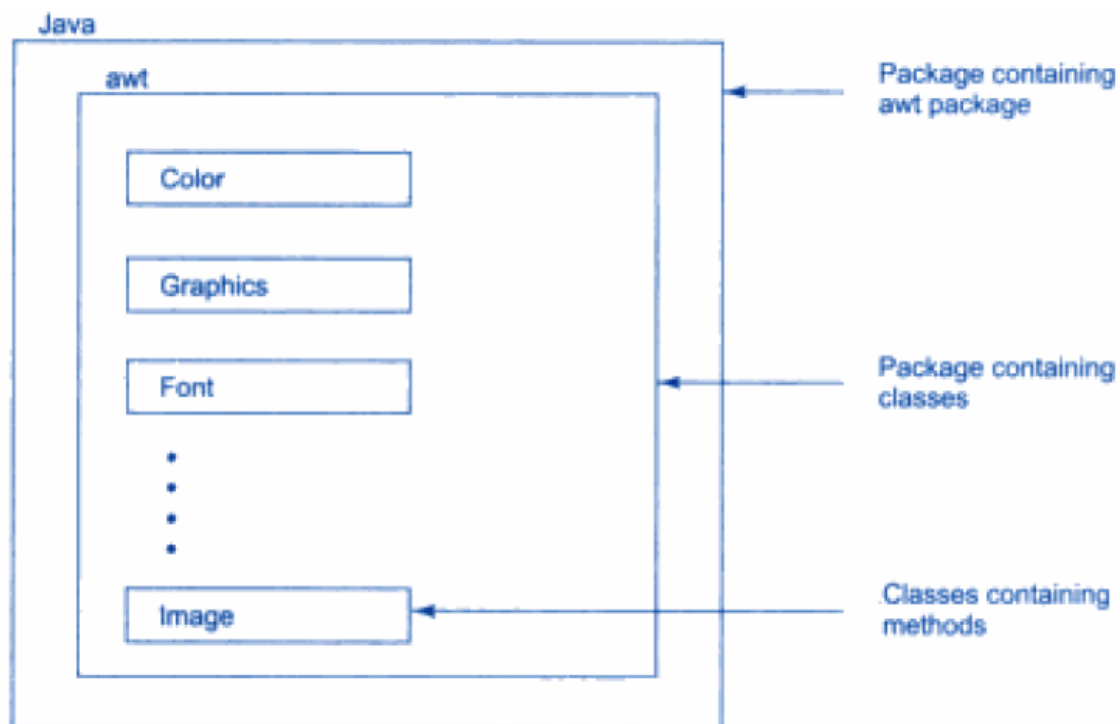


**Fig. Java API Packages**

Java System Packages and Their Classes	
java.lang	Language support classes. These are classes that Java compiler itself uses and therefore they are automatically imported. They include classes for primitive types, strings, math functions, threads and exceptions
java.util	Language utility classes such as vectors, hash tables, random numbers, date, etc.
java.io	Input/output support classes. They provide facilities for the input and output of data.
java.awt	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.
java.net	Classes for networking. They include classes for communicating with local computers as well as with internet servers.
java.applet	Classes for creating and implementing applets.

### Using System Packages:

The packages are organized in a hierarchical structure. This shows that the package named java contains the package awt, which in turn contains various classes required for implementing graphical user interface.



There are two ways of accessing the classes stored in a package. The first approach is to use the fully qualified class name of the class that we want to use. This is done by using the package

name containing the class and then appending the class name to it using the dot operator. For example, if we want to refer to the class Color in the awt package, then we may do so as follows:

**java.awt.Color**

### Creating Packages:

In java to create our own packages. We must first declare the name of the package using the package keyword followed by a package name. This must be the first statement in a Java source file (except for comments and white spaces). Then we define a class, just as we normally define a class.

**Here is an example:**

```
package firstPackage; // package declaration
public class FirstClass; // class definition

{
    -----
    ----- (body of class)
    -----
}
```

Here the package name is firstPackage. The class FirstClass is now considered a part of this package. This listing would be saved as a file called FirstClass.java, and located in a directory named first Package.

When the source file is compiled, Java will create a class file and store it in the same directory. Remember that the .class files must be located in a directory that has the same name as the package, and this directory should be a subdirectory of the directory where classes that will import the package are located.

To create package involves the following steps:

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

## Accessing a Package:

Java system package be accessed either using a fully qualified class name or using a shortcut approach through the import statement. We use the import statement when there are many references to a particular package or the package name is too long and unwieldy.

The same approaches can be used to access the user-defined packages as well. The import statement can be used to search a list of packages for a particular class. The general form of import statement for searching a class is as follows:

```
import package1 [.package2] [.package3].classname;
```

Here package1 is the name of the top level package, package2 is the name of the package that is inside the package1, and so on. We can have any number of packages in a package hierarchy. Finally, the explicit classname is specified. Note that the statement must end with a semicolon (;).

Example of importing a particular class:

```
import firstPackage.secondPackage.MyClass;
```

After defining this statement, all the members of the class MyClass can be directly accessed using the class name or its objects (as the case may be) directly without using the package name.

We can also use another approach as follows:

```
import packagename.*;
```

The star (\*) indicates that bring all classes. This implies that we can access all classes contained in the above package directly.

## Using a Package:

Let us now consider some simple programs that will use classes from other packages. The listing below shows a package named package1 containing a single class ClassA.

```
package package1;  
public class ClassA  
{  
    public void displayA()  
    {  
        System.out.println("Class A");  
    }  
}
```

```
    }  
}
```

This source file should be named ClassA.java and stored in the subdirectory. Now compile this java file. The resultant ClassA.class will be stored in the same subdirectory.

**Example: Creating and accessing package.**

-----

## Managing Errors and Exceptions:

### Introduction:

Rarely does a program run successfully at its very first attempt. It is common to make mistakes while developing as well as typing a program. A mistake might lead to an error causing to program to produce unexpected results. Errors are the wrongs that can make a program go wrong. An error may produce an incorrect output or may terminate the execution of the program or even may cause the system to crash. It is therefore important to detect and manage properly all the possible error conditions in the program so that the program will not terminate or crash during execution.

### Types of Errors:

Errors may broadly be classified into two categories:

- 1) Compile-time errors
- 2) Run-time errors

#### 1) Compile-Time Errors:

All syntax errors will be detected and displayed by the Java compiler and therefore these errors are known as compile-time errors. Whenever the compiler displays an error, it will not create the class file. It is therefore necessary that we fix all the errors before we can successfully compile and run the program.

#### Example Program:

```
/* This program contains an error */  
class Error1  
{  
    public static void main(String args[])  
    {  
        System.out.println("Hello Java!") // Missing;
```

```
}  
}
```

The Java compiler does a nice job of telling us where the errors are in the program. For example, if we have missed the semicolon at the end of print statement in Program the following message will be displayed in the screen:

```
Error1.java:7: ';' expected  
System.out.println ("Hello Java!")  
1 error
```

- Most of the compile-time errors are due to typing mistakes.
- Missing semicolons
- Missing (or mismatch of) brackets in classes and methods
- Misspelling of identifiers and keywords
- Missing double quotes in strings
- Use of undeclared variables
- Incompatible types in assignments/ initialization
- Bad references to objects • Use of in place of operator
- And so on

Other errors we may encounter are related to directory paths. An error such as `javac: command not found` means that we have not set the path correctly.

## 2) Run-Time Errors:

Sometimes, a program may compile successfully creating the .class file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow. Most common run-time errors are:

- Dividing an integer by zero
- Accessing an element that is out of the bounds of an array
- Trying to store a value into an array of an incompatible class or type
- Trying to cast an instance of a class to one of its subclasses
- Passing a parameter that is not in a valid range or value for a method.
- Trying to illegally change the state of a thread
- Attempting to use a negative size for an array
- Using a null object reference as a legitimate object
- Converting invalid string to a number.



- Accessing a character that is out of bounds of a string
- And may more

When such errors are encountered, Java typically generates an error message program. Program illustrates how a run-time error causes termination of execution of the program.

**Program Illustration of run-time errors:**

```
class Error2
{
    public static void main(String args[])
    {
        int a=10;
        int b = 5;
        int c=5;
        int x=a/(b-c); // Division by zero
        System.out.println("x="+x);
        int y=a/(b+c);
        System.out.println("y=" + y);
    }
}
```

**Exceptions:**

An exception is a condition that is caused by a run-time error in the program. When the Java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws it (i.e. informs us that an error has occurred). If the exception object is not caught and handled properly, the interpreter will display an error message. If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as exception handling. The mechanism suggests incorporation of a separate error handling code that performs the following tasks:

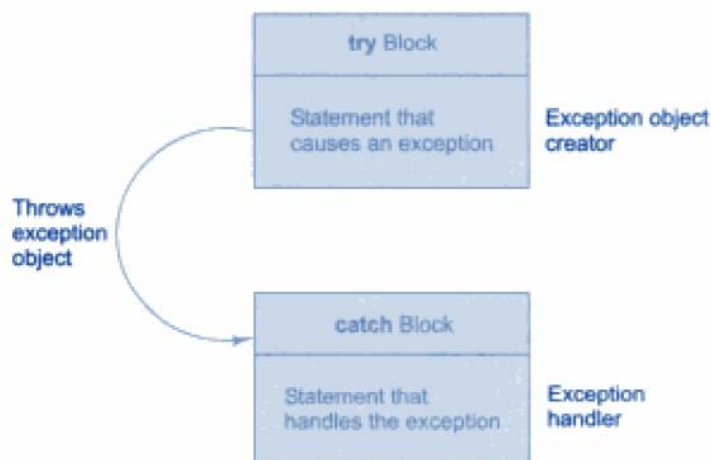
1. Find the problem (Hit the exception).
2. Inform that an error has occurred (Throw the exception)
3. Receive the error information (Catch the exception)
4. Take corrective actions (Handle the exception)

The error handling code basically consists of two segments, one to detect errors and to throw exceptions and the other to catch exceptions and to take appropriate actions. Some common exceptions that we must watch out for catching are listed below

Exception Type	Cause of Exception
ArithmeticException	Caused by math errors such as division by zero
ArrayOutOfBoundsException	Caused by bad array indexes
ArrayStoreException	Caused when a program tries to store the wrong type of data in an array
FileNotFoundException	Caused by an attempt to access a nonexistent file
IOException	Caused by general I/O failures, such as inability to read from a file
NullPointerException	Caused by referencing a null object.
NumberFormatException	Caused when a conversion between strings and number fails
OutOfMemoryException	Caused when there's not enough memory to allocate a new object
SecurityException	Caused when an applet tries to perform an action not allowed by the browser's security setting
StackOverflowException	Caused when the system runs out of stack space.
StringOutOfBoundsException	Caused when a program attempts to access a nonexistent character position in a string

### Syntax of Exception Handling Code:

The basic concepts of exception handling are throwing an exception and catching it.



**Fig. 13.1**

*Exception handling mechanism*

Java uses a keyword `try` to preface a block of code that is likely to cause an error condition and "throw" an exception. A catch block defined by the keyword `catch` "catches" the exception "thrown" by the try block and handles it appropriately. The catch block is added immediately after the try block. The following example illustrates the use of simple try and catch statements:

```
try
{
    statement; // generates an exception
}
catch (Exception-type e)
{
    statement; // processes the exception
}
```

The try block can have one or more statements that could generate an exception. If anyone statement generates an exception, the remaining statements in the block are skipped and execution jumps to the catch block that is placed next to the try block.

The catch block too can have one or more statements that are necessary to process the exception. Remember that every try statement should be followed by at least one catch statement; otherwise compilation error will occur.

Note that the catch statement works like a method definition. The catch statement is passed a single parameter, which is reference to the exception object thrown (by the try block). If the catch parameter matches with the type of exception object, then the exception is caught and statements in the catch block will be executed. Otherwise, the exception is not caught and the default exception handler will cause the execution to terminate.

### **Program: Using try and catch for exception handling**

```
class Error3
{
    public static void main(String args[])
    {
        int a = 10;
        int b = 5;
        int c = 5;
        int x,y;
        try
        {
```

```
        x = a/ (b-c);
    }
    catch (ArithmeticException e)
    {
        System.out.println("Division by zero");
    }
    y = a / (b+c);
    System.out.println("y =" + y);
}
}
```

**Program: Catching invalid command line arguments**

```
class CLineInput
{
    public static void main(String args[])
    {
        int invalid = 0; // Number of invalid arguments
        int number, count = 0;
        for (int i = 0; i < args.length; i++)
        {
            try
            {
                number = Integer.parseInt(args[i]);
            }
            catch(NumberFormatException e)
            {
                invalid=invalid + 1; // Caught an invalid number
                System.out.println("Invalid Number : "+args[i]);
                continue; // Skip the remaining part of the loop
            }
            count = count + 1;
        }
        System.out.println("Valid Numbers = " + count);
        System.out.println("Invalid Numbers = "+ invalid);
    }
}
```

## Multiple Catch Statements

It is possible to have more than one catch statement in the catch block as illustrated below.

```
try
{
    Statement; // generates an exception
}
catch (Exception-Type-1 e)
{
    Statement; // processes exception type 1
}
catch (Exception-Type-2 e)
{
    Statement; // processes exception type 2
}
catch (Exception-Type-N e)
{
    Statement; // processes exception type N
}
```

### Program: Using multiple catch blocks

```
class Erro4
{
    public static void main(String args[])
    {
        int a[] = {5, 10};
        int b = 5;
        try
        {
            int x = a[2] / b - a[1];
        }
        catch (ArithmeticException e)
        {
            System.out.println("Division by zero");
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Array index error");
        }
        catch (ArrayStoreException e)
        {
        }
    }
}
```

```
        {  
            System.out.println("wrong data type");  
        }  
        int y = a[1] / a[0];  
        System.out.println("y = " + y);  
    }  
}
```

### Using Finally Statement:

Java supports another statement known as finally statement that can be used to handle an exception that is not caught by any of the previous catch statements, finally block can be used to handle any exception generated within a try block. It may be added immediately after the try block or after the last catch block shown as follows:

```
try  
{  
    -----  
    -----  
}  
finally  
{  
    -----  
    -----  
}
```

When a finally block is defined, this is guaranteed to execute, regardless of whether or not an exception is thrown.

In Program, we may include the last two statements inside a finally block as shown below:

```
finally  
{  
    int y=a[1]/a[0];  
    System.out.println("y="+y);  
}
```

### Throwing Our Own Exceptions:

There may be times when we would like to throw our own exceptions. We can do this by using the keyword throw as follows:

**throw new Throwable\_subclass;**

Examples:

**throw new ArithmeticException();**

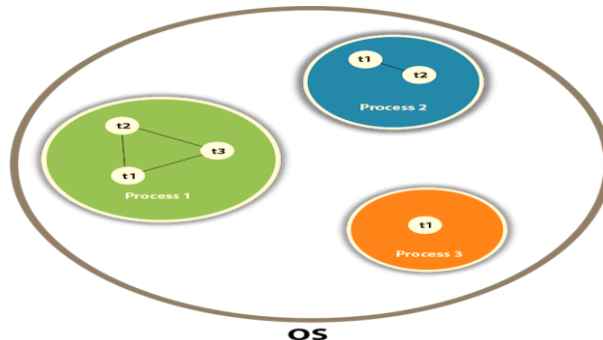
**throw new Numberformat Exception();**

### **Program: Throwing our own exception**

```
import java.lang.Exception;
class MyException extends Exception
{
    MyException(String message)
    {
        super (message);
    }
}
class TestMyException
{
    public static void main(String args[])
    {
        int x = 5, y = 1000;
        try
        {
            float z =(float) x/(float) y;
            if(z < 0.01)
            {
                throw new MyException("Number is too small");
            }
        }
        catch (MyException e)
        {
            System.out.println("Caught my exception");
            System.out.println(e.getMessage());
        }
        finally
        {
            System.out.println("I am always here");
        }
    }
}
```

## Multithreading:

**Multithreading** in Java is a process of executing multiple threads simultaneously. Multithreading is useful in a number of ways. It enables programmers to do multiple things at one time. They can divide a long program (containing operations that are conceptually concurrent) into threads and execute them in parallel. Multithreading approach would considerably improve the speed of our programs.



As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

**Thread-** A thread is a lightweight sub-process, the smallest unit of processing.

**Creating thread:** Creating thread in java is simple. We can create thread in two ways

1. Extending thread class
2. Implementing runnable interface.

### 1. Extending thread class:

1. Declare the new class that extends thread class.

```
class mythread extends Thread
{
    .....
    .....
    .....
}
```

2. Implement the run method. Extending class must override the run method to define the code executed by thread.

```
public void run()
{
```



```
}
```

3. Create thread object and call the start() method for execution of a thread.

```
mythread t1 =new mythread ();
```

```
t1.start();
```

**Example:**

```
class A extends Thread
{
    public void run()
    {
        for(int i=1;i<=5;i++)
        {
            System.out.println("Thread A: i="+i);
        }
    }
}
class B extends Thread
{
    public void run()
    {
        for(int j=1;j<=5;j++)
        {
            System.out.println("Thread B: j="+j);
        }
    }
}
class test
{
    public static void main (String args[])
    {
        A t1=new A();
        B t2=new B();
        t1.start();
        t2.start();
    }
}
```

## 2. Implementing Runnable Interface

1. Declare the class that implements runnable interface.
2. Implement run () method.
3. Create thread by defining an object that instantiated from this runnable class.
4. Call thread's start () method to run the thread.

### Example

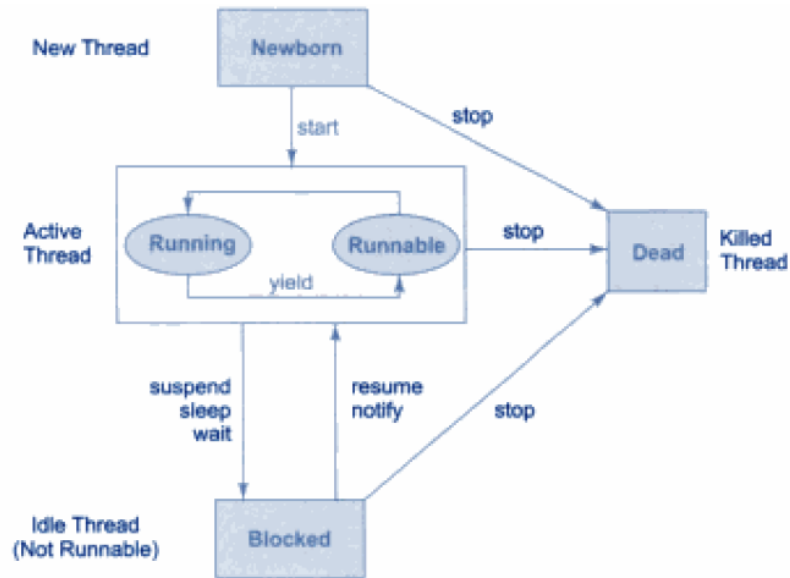
```
class A implements Runnable
{
    public void run()
    {
        for(int i=1;i<=5;i++)
        {
            System.out.println("Thread A: i="+i);
        }
    }
}
class test
{
    public static void main (String args[])
    {
        A t1=new A();
        Thread t2=new Thread(t1);
        t2.start();
    }
}
```

### Life Cycle of a Thread:

During the life time of a thread, there are many states it can enter. They include:

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

A thread is always in one of these five states. It can move from one state to another via a variety of ways as shown in Fig.

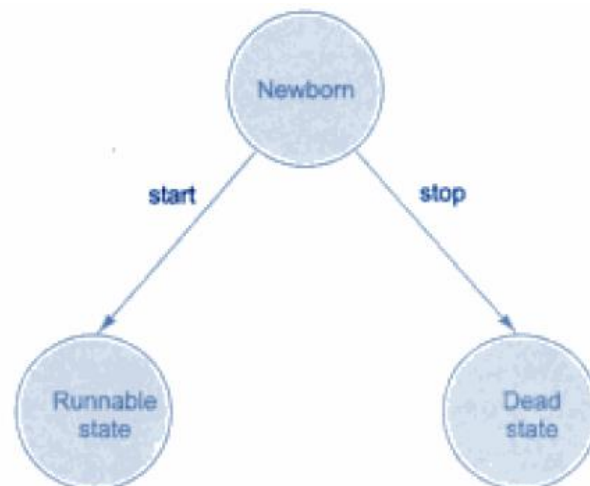


**Fig. 12.3** State transition diagram of a thread

### 1. Newborn State:

When we create a thread object, the thread is born and is said to be in newborn state. The thread is not yet scheduled for running. At this state, we can do only one of the following things with it:

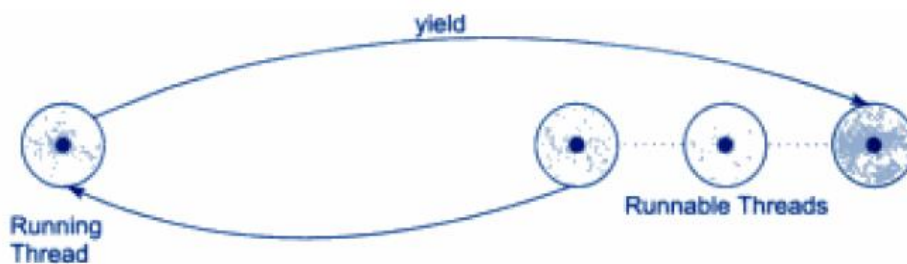
- Schedule it for running using start() method.
- Kill it using stop() method.



**Fig. 12.4** Scheduling a newborn thread

## 2. Runnable state:

The runnable state means that the thread is ready for execution and is waiting for the availability of the processor. That is, the thread has joined the queue of threads that are waiting for execution. If all threads have equal priority, then they are given time slots for execution in round robin fashion, i.e., first-come, first-serve manner. The thread that relinquishes control joins the queue at the end and again waits for its turn. This process of assigning time to threads is known as time-slicing. However, if we want a thread to relinquish control to another thread to equal priority before its turn comes, we can do so by using the `yield()` method.



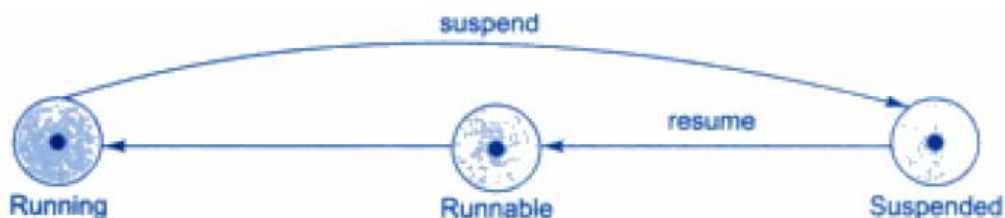
**Fig. 12.5** Relinquishing control using `yield()` method

## 3. Running State:

Running means that the processor has given its time to the thread for its execution. The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread.

A running thread may relinquish its control in one of the following situations.

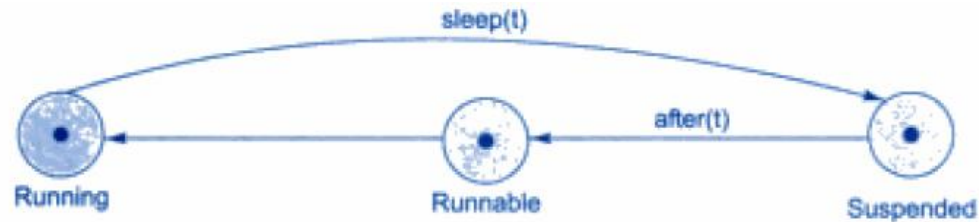
- 1) It has been suspended using `suspend()` method. A suspended thread can be revived by using the `resume()` method. This approach is useful when we want to suspend a thread for some time due to certain reason, but do not want to kill it.



**Fig. 12.6** Relinquishing control using `suspend()` method

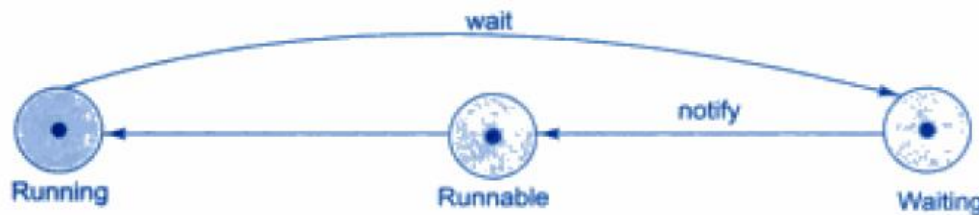
- 2) It has been made to sleep. We can put a thread to sleep for a specified time period using the method `sleep(time)` where time is in milliseconds. This means that the thread is out of

the queue during this time period. The thread re-enters the runnable state as soon as this time period is elapsed.



**Fig. 12.7** Relinquishing control using `sleep()` method

- 3) It has been told to wait until some event occurs. This is done using the `wait()` method. The thread can be scheduled to run again using the `notify()` method.



**Fig. 12.8** Relinquishing control using `wait()` method

#### 4. Blocked State:

A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running state. This happens when the thread is suspended, sleeping, or waiting in order to satisfy certain requirements. A blocked thread is considered "not runnable" but not dead and therefore fully qualified to run again.

#### 5. Dead State:

Every thread has a life cycle. A running thread ends its life when it has completed executing its `run()` method. It is a natural death.

### Thread Priority:

Every thread in Java has a priority that helps the thread scheduler to determine the order in which threads are scheduled. The thread with higher priority will be usually run before the lower priority threads. The threads of the same priority will run "first come first serve" basis.

Java permits us to set the priority of thread using the `setPriority()` method as follows

**Threadname.setPriority(int number)**

The **int number** is an integer value that must be between 1 to 10 to set thread priority. Thread defines several priority constants.

- MIN\_PRIORITY = 1
- MAX\_PRIORITY = 10
- NORM\_PRIORITY = 5

The default priority is 5 that is NORM\_PRIORITY. If you specify a priority out of range then IllegalArgumentException will be thrown.

**Example:**

```
import java.io.*
class A extends Thread
{
    public void run()
    {
        for(int i=1;i<=5;i++)
        {
            System.out.println("Thread A: i="+i);
        }
        System.out.println("Exit From thread A");
    }
}
class B extends Thread
{
    public void run()
    {
        for(int j=1;j<=5;j++)
        {
            System.out.println("Thread B: j="+j);
        }
        System.out.println("Exit From thread B");
    }
}
class C extends Thread
{
    public void run()
    {
        for(int k=1;k<=5;k++)
        {
```

```
        System.out.println("Thread C: k="+k);
    }
    System.out.println("Exit From thread C");
}
}
class test
{
    public static void main(String args[])
    {
        A t1=new A();
        B t2=new B();
        C t3=new C();
        t1.setPriority(4);
        t2.setPriority(1);
        t3.setPriority(7);
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Output:

```
Thread C: k=1
Thread C: k=2
Thread C: k=3
Thread C: k=4
Thread C: k=5
Exit From thread C
Thread A: i=1
Thread A: i=2
Thread A: i=3
Thread A: i=4
Thread A: i=5
Exit From thread A
Thread B: j=1
Thread B: j=2
Thread B: j=3
Thread B: j=4
Thread B: j=5
Exit From thread B
```

### **Synchronization:**

- Synchronization is the process of controlling access of shared resources by multiple threads.
- The main purpose of synchronization is to avoid thread interference.
- At times when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.

**Syntax**

```
Synchronized returntype methodname()  
{  
    ----- //code here is synchronized  
}
```

**Why we need Synchronization?**

If we do not use synchronization, and let two or more threads access a shared resource at the same time, it will lead to inconsistent results.

Consider an example, suppose we have two different threads T1 and T2, T1 starts execution and save certain values in a file *temporary.txt* which will be used to calculate some result when T1 returns. Meanwhile, T2 starts and before T1 returns, T2 change the values saved by T1 in the file *temporary.txt* (*temporary.txt* is the shared resource). Now obviously T1 will return wrong result.

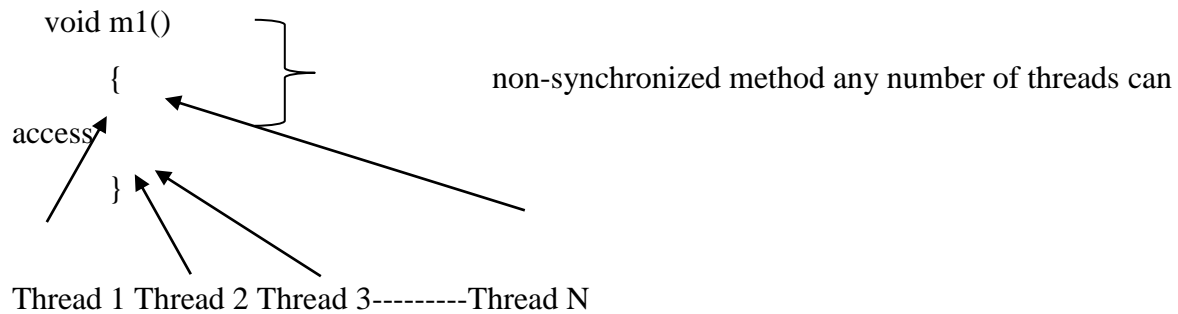
To prevent such problems, synchronization was introduced. With synchronization in above case, once T1 starts using *temporary.txt* file, this file will be locked(LOCK mode), and no other thread will be able to access or modify it until T1 returns.

**Synchronization is achieved through synchronized methods.**

- If a method or a block declared as synchronized then at a time only one Thread is allowed to operate on the given object.
- The main advantage of synchronization is, we can resolve data inconsistency problems.



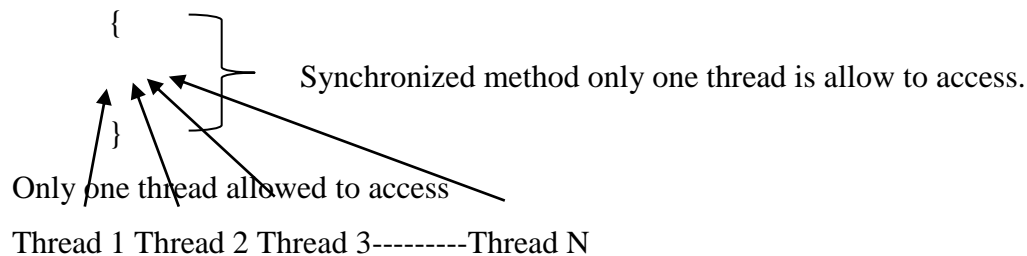
## Non-synchronized methods:



- 1) In the above case multiple threads are accessing the same methods hence we are getting data inconsistency problems. These methods are not thread safe methods.
- 2) But in this case multiple threads are executing so the performance of the application will be increased.

## Synchronized methods:

synchronized void m2()



- 1 In the above case only one thread is allow to operate on particular method so the data inconsistency problems will be reduced.
- 2 Only one thread is allowed to access so the performance of the application will be reduced.
- 3 If we are using above approach there is no multithreading concept.

class Test

```
{
    public static synchronized void x(String msg) //only one thread is able to access
    {
        try
        {
            System.out.println(msg);
            Thread.sleep(4000);
            System.out.println(msg);
            Thread.sleep(4000);
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```

```
    }
    }
}
class MyThread1 extends Thread
{
    public void run( )
    {
        Test.x("ratan");
    }
}
class MyThread2 extends Thread
{
    public void run( )
    {
        Test.x("anu");
    }
}
class MyThread3 extends Thread
{
    public void run( )
    {
        Test.x("banu");
    }
}
class TestDemo
{
    public static void main(String[] args)    //main thread -1
    {
        MyThread1 t1 = new MyThread1();
        MyThread2 t2 = new MyThread2();
        MyThread3 t3 = new MyThread3();
        t1.start();    //2-Threads
        t2.start();    //3-Threads
        t3.start(); //4-Threads
    }
}
```

Output

**If method is synchronized:**

C:\j2sdk1.4.2\_13\bin>javac TestDemo.java

C:\j2sdk1.4.2\_13\bin>java TestDemo

ratan

ratan

anu

anu

banu

banu

**if method is non-synchronized:-**

```
C:\j2sdk1.4.2_13\bin>javac TestDemo.java
```

```
C:\j2sdk1.4.2_13\bin>java TestDemobanu
```

```
ratana
```

```
anubanu
```

```
banu
```

```
anuratan
```

**Inter Thread-communication:**

Inter thread communication is a mechanism in which a thread releases the lock and enter into paused state and another thread acquires the lock and continue to executed. it is implemented by following methods of object class.

**1. wait():** if any thread calls the wait() method then it causes the current thread to release the lock and wait until another thread invokes the notify() or notifyAll() method for this object or a specified amount of time has elapsed.

**Syntax:**

```
public final void wait() throws InterruptedException
```

```
public final void wait(long timeout) throws InterruptedException
```

**2. notify ():** this method is used **to wake up a single thread** and releases the object lock.

**Syntax:**

```
public final void notify()
```

**3. notifyAll ():** this method is used **to wake up all threads** that are in waiting state. This method gives the notification to **all waiting threads** of a particular object. Which one thread will wake up first depends on thread priority and OS implementation.

**Syntax:**

```
public final void notifyAll()
```

**Note-** to call wait(), notify(), or notifyAll() method on any object, thread should own the lock of that object i.e. the thread should be inside synchronized area.

## UNIT NO 4

### AWT, SWING (JFC)

#### Introduction & Components of AWT

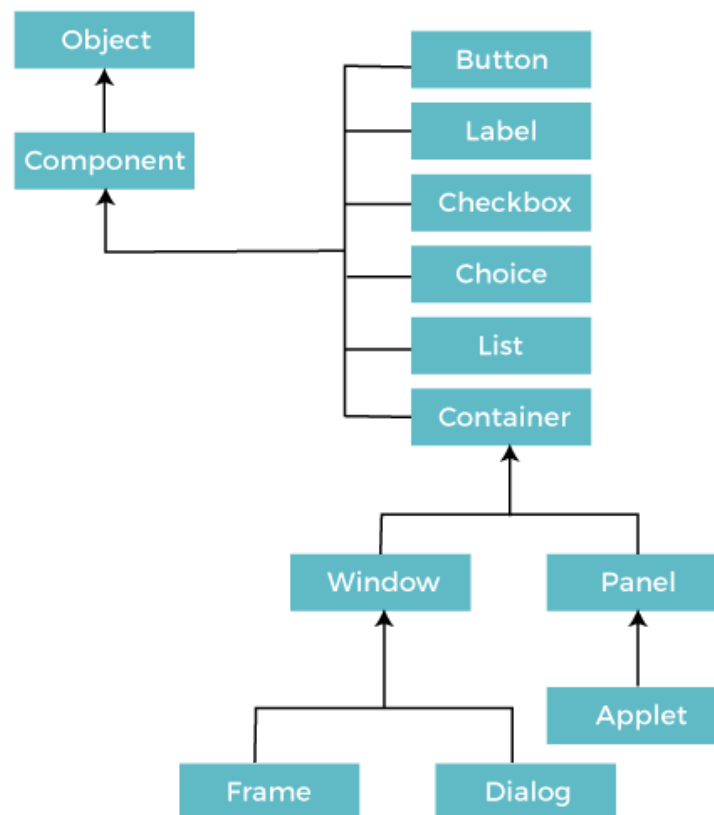
Java AWT (Abstract Window Toolkit) is an API to develop Graphical User Interface (GUI) or windows-based applications in Java.

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavy weight i.e. its components are using the resources of underlying operating system (OS).

Java AWT was developed by Sun Microsystems In 1995. It is heavy-weight in use because it is generated by the system's host operating system. It contains a large number of classes and methods, which are used for creating and managing GUI.

The java.awt package provides classes for AWT API such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

**Java AWT Hierarchy:** The hierarchy of Java AWT classes are given below.



## AWT Components:

All the elements like the button, text fields, scroll bars, etc. are called components. In Java AWT, there are classes for each component as shown in above diagram. In order to place every component in a particular position on a screen, we need to add them to a container.

## Container:

The Container is a component in AWT that can contain another components like buttons , textfields, labels etc. The classes that extends Container class are known as container such as **Frame**, **Dialog** and **Panel**. It is basically a screen where the components are placed at their specific locations. Thus it contains and controls the layout of components.

## Types of containers:

There are four types of containers in Java AWT:

- 1) Window
- 2) Panel
- 3) Frame
- 4) Dialog

- 1) **Window:** The window is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window. We need to create an instance of Window class to create this container.
- 2) **Panel:** The Panel is the container that doesn't contain title bar, border or menu bar. It is generic container for holding the components. It can have other components like button, text field etc. An instance of Panel class creates a container, in which we can add components.
- 3) **Frame:** The Frame is the container that contain title bar and border and can have menu bars. It can have other components like button, text field, scrollbar etc. Frame is most widely used container while developing an AWT application.

### 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.
--	--

## Java AWT Example:

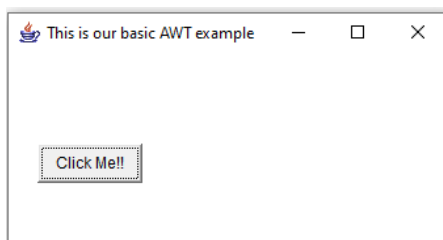
To create simple AWT example, you need a frame. There are two ways to create a GUI using Frame in AWT.

- By extending Frame class (inheritance)
- By creating the object of Frame class (association)

**i) AWT Example by Inheritance:** Let's see a simple example of AWT where we are inheriting Frame class. Here, we are showing Button component on the Frame.

```
import java.awt.*; // importing Java AWT class
// extending Frame class to our class AWTEExample1
public class AWTEExample1 extends Frame
{
    AWTEExample1()
    {
        Button b = new Button("Click Me!!"); // creating a button
        b.setBounds(30,100,80,30); // setting button position on screen
        add(b); // adding button into frame
        setSize(300,300); // frame size 300 width and 300 height
        setTitle("This is our basic AWT example"); // setting the title of Frame
        setLayout(null); // no layout manager
        setVisible(true); // now frame will be visible, by default it is not visible
    }
    public static void main(String args[])
    {
        // creating instance of Frame class
        AWTEExample1 f = new AWTEExample1();
    }
}
```

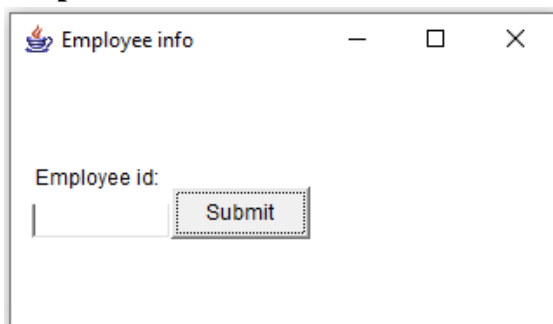
Output:



- ii) **AWT Example by Association:** Let's see a simple example of AWT where we are creating instance of Frame class. Here, we are creating a TextField, Label and Button component on the Frame.

```
import java.awt.*; // importing Java AWT class
// class AWTEExample2 directly creates instance of Frame class
class AWTEExample2
{
    // initializing using constructor
    AWTEExample2()
    {
        Frame f = new Frame(); // creating a Frame
        Label l = new Label("Employee id:"); // creating a Label
        Button b = new Button("Submit"); // creating a Button
        TextField t = new TextField(); // creating a TextField
        l.setBounds(20, 80, 80, 30); // setting position of above components in the frame
        t.setBounds(20, 100, 80, 30);
        b.setBounds(100, 100, 80, 30);
        f.add(b); // adding components into frame
        f.add(l);
        f.add(t);
        f.setSize(400,300); // frame size 300 width and 300 height
        f.setTitle("Employee info"); // setting the title of frame
        f.setLayout(null);
        f.setVisible(true); // setting visibility of frame
    }
    public static void main(String args[])
    {
        AWTEExample2 obj = new AWTEExample2(); // creating instance of Frame class
    }
}
```

**Output:**



## Event Delegation Model:

In Event Delegation Model a source creates an event and sends it to one or more listeners. The listener accepts that event. Once the event is accepted, the listener handles that event using a separate piece of code. The advantage of this Model is that the logic which handles events is completely separated from the user interface logic that generates those events. The following sections define different terms used in Delegation Event Model.

**1. Event:** An event is an action which is generated by source. For e.g. when we click on button it should proceed further, so clicking is an event.

**2. Event Source:** A source is an object which generates an event. In above example button is source of event.

Following table gives different sources of events.

Event Source	Description
Button	Generates action events when the button is pressed.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Check box	Generates item events when the check box is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, opened, or quit.
Keys on keyboard	Generates key Event when key is pressed
Mouse	Generates mouse Event when mouse is pressed

**3. Event Listener:** A listener is an interface which is notified when an event is generated. It must be registered with one or more sources to receive notification about events. It also implements methods to process events.

**4. Event Classes:** There are different Event classes available for e.g. ActionEvent class for handling button, list, Menubar related events. Following table provides different Event Classes.



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; also 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.
MouseWheelEvent	Generated when the mouse wheel moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when window is activated, closed, deactivated, opened or quite.

## Steps to perform Event Handling:

Following steps are required to perform event handling:

1. Register the component with the Listener

**2) Registration Methods:** For registering the component with the Listener, many classes provide the registration methods. For example:

- **Button**
  - `public void addActionListener(ActionListener a){ }`
- **MenuItem**
  - `public void addActionListener(ActionListener a){ }`
- **TextField**
  - `public void addActionListener(ActionListener a){ }`
  - `public void addTextListener(TextListener a){ }`
- **TextArea**
  - `public void addTextListener(TextListener a){ }`
- **Checkbox**
  - `public void addItemListener(ItemListener a){ }`

- **Choice**
  - `public void addItemListener(ItemListener a){ }`
- **List**
  - `public void addActionListener(ActionListener a){ }`
  - `public void addItemListener(ItemListener a){ }`

## Java Event Handling Code:

We can put the event handling code into one of the following places:

1. Within class
2. Other class
3. Anonymous class

### Example: Within Class

```
import java.awt.*;
import java.awt.event.*;
class AEvent extends Frame implements ActionListener
{
    TextField tf;
    AEvent()
    {
        //create components
        tf=new TextField();
        tf.setBounds(60,50,170,20);
        Button b=new Button("click me");
        b.setBounds(100,120,80,30);
        //register listener
        b.addActionListener(this);//passing current instance
        //add components and set size, layout and visibility
        add(b);
        add(tf);
        setSize(300,300);
        setLayout(null);
        setVisible(true);
    }
    public void actionPerformed(ActionEvent e)
    {
        tf.setText("Welcome");
    }
    public static void main(String args[])
    {
```

```

        new AEvent();
    }
}

```

## AWT Components:

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. The java.awt package provides classes for AWT API such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

- 1) AWT Label:** The easiest control to use is a label. A label contains a string and label is an object of type Label class. Labels are passive controls as it does not create any event when it is accessed. It is used to display a single line of **read only text**. The text can be changed by a programmer but a user cannot edit it directly.

### AWT Label Class Declaration

`public class Label extends Component implements Accessible`

### Label Class Constructors:

Sr. No.	Constructor	Description
1.	Label()	It creates an empty label.
2.	Label(String text)	It creates a label with the given string (left justified by default).
3.	Label(String text, int alignment)	It creates a label with the specified string and the specified alignment. Value of alignment must one of the following 3 constant <b>Label.LEFT</b> , <b>Label.RIGHT</b> , <b>Label.CENTER</b>

### Label Methods:

Sr. No.	Method name	Description
1.	void setText(String text)	It is used to sets the texts for label with the specified text.
2.	void setAlignment(int alignment)	It is used to sets the alignment for label with the specified alignment.
3.	String getText()	It is used to get the text of the label

4.	int getAlignment()	It gets the current alignment of the label.
----	--------------------	---

**Creating Label:**

```
Label l=new Label();
```

```
Label l = new Label(String);
```

**Example:**

```
import java.awt.*;

public class LabelExample
{
    public static void main(String args[])
    {
        //creating the object of Frame class and Label class
        Frame f = new Frame ("Label example");
        // Creating the labels
        Label lbl1 = new Label ("First Label.");
        Label lbl2 = new Label ("Second Label.");
        // set the location of label
        lbl1.setBounds(50, 100, 100, 30);
        lbl2.setBounds(50, 150, 100, 30);
        // adding labels to the frame
        f.add(lbl1);
        f.add(lbl2);
        // setting size, layout and visibility of frame
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```

**Output**



- 2) **AWT Button:** The most widely used control is Button. A push button is a component that contains a label and generates an event when it is pressed.

**Constructors:**

Button ( ) -> Creates a empty button.

Button ( String Str ) -> Creates a button that contains Str as a label.

**Syntax:**

Button b1 = new Button ( "Text" );

**Example:**

Button b1 = new Button ( "cancel" );

This creates a button with label "cancel" .

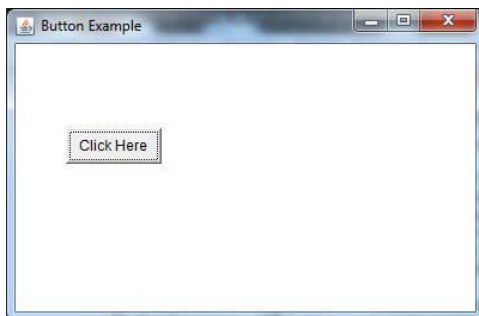
**Methods:**

Sr. No.	Method	Description
1.	void setText (String text)	It sets the string message on the button
2.	String getText()	It fetches the String message on the button.
3.	void setLabel (String label)	It sets the label of button with the specified string.
4.	String getLabel()	It fetches the label of the button.
5	setLocation (int x , int y )	It sets the location of button. (x, y ) denotes the co-ordinates of the top left corner of the button.
6	void setsize ( int w, int h )	It sets the size of button (w, h ) denotes the width and height of the button.
7	void setbounds ( int x, int y, int w, int h )	It is used to set the location and size of button on frame. ( x, y ) -> coordinates of the top left corner. ( w, h ) -> width and height of button in pixels

**Example:**

```
import java.awt.*;

public class ButtonExample
{
    public static void main (String[] args)
    {
        // create instance of frame with the label
        Frame f = new Frame("Button Example");
        // create instance of button with label
        Button b = new Button("Click Here");
        // set the position for the button in frame
        b.setBounds(50,100,80,30);
        // add button to the frame
        f.add(b);
        // set size, layout and visibility of frame
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```

**Output:****3) AWT Checkbox:**

The Checkbox class is used to create a checkbox. It is used to turn an option on (true) or off (false). Clicking on a Checkbox changes its state from "on" to "off" or from "off" to "on". 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 checkbox represents. We change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group.

### AWT Checkbox Class Declaration

`public class` Checkbox `extends` Component `implements` ItemSelectable, Accessible

### Checkbox Class Constructors:

Sr. No.	Constructor	Description
1.	Checkbox()	It constructs a checkbox with no string as the label.
2.	Checkbox(String label)	It constructs a checkbox with the given label.
3.	Checkbox(String label, boolean state)	It constructs a checkbox with the given label and sets the given state.(i.e. true or false)

### Methods:

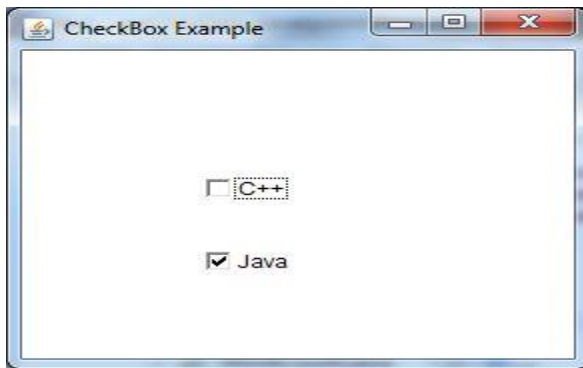
Sr. No.	Method	Description
1.	String getLabel()	It is used to get the label of the checkbox.
2.	void setLabel(String label)	It is used to assign label for the checkbox.
3.	void setState(boolean state)	It is used to assign state for the checkbox.
4.	boolean getState()	It returns true if the checkbox is on, else returns off.

### Example:

```
import java.awt.*;

public class checkboxExample
{
    public static void main (String[] args)
    {
        // create instance of frame with the label
        Frame f = new Frame("Checkbox Example");
        // create instance of button with label
        Checkbox c = new Checkbox ("C++");
        Checkbox c1 = new Checkbox ("Java", true);
        // set the position for the button in frame
        c.setBounds(100,100,80,30);
```

```
c.setBounds(100,150,80,30);  
// add button to the frame  
f.add(c);  
f.add(c1);  
// set size, layout and visibility of frame  
f.setSize(400,400);  
f.setLayout(null);  
f.setVisible(true);  
}  
}
```

**Output:****4) AWT CheckboxGroup: Radio Button**

In Java, AWT contains a `CheckboxGroup` Class. It is used to group a set of `Checkbox`. When `Checkboxes` are grouped then only one box can be checked at a time.

**CheckboxGroup Class Declaration:**

```
public class CheckboxGroup extends Object implements Serializable
```

**Creating Radiobutton:**

```
CheckboxGroup cbg = new CheckboxGroup();
```

```
Checkbox rb = new Checkbox(Label, cbg, boolean);
```

**Example:**

This example creates a `checkboxgroup` that is used to group multiple `checkbox` in a single unit. It is helpful when we have to select single choice among the multiples.

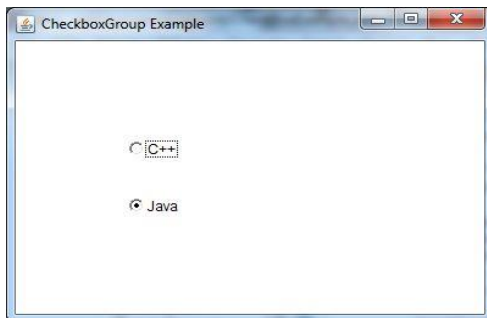
```
import java.awt.*;
```

```
public class checkboxgroupExample
```

```
{
```



```
public static void main(String args[])
{
    Frame f= new Frame("CheckboxGroup Example");
    CheckboxGroup cbg = new CheckboxGroup();
    Checkbox checkBox1 = new Checkbox("C++", cbg, false);
    checkBox1.setBounds(100,100, 50,50);
    Checkbox checkBox2 = new Checkbox("Java", cbg, true);
    checkBox2.setBounds(100,150, 50,50);
    f.add(checkBox1);
    f.add(checkBox2);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
}
```

**Output:****Swing (JFC):**

Swing is a part of **JFC (Java Foundation Classes)**. Building Graphical User Interface in Java requires the use of Swings. Swing Framework contain a large set of components which allow high level of customization and provide rich functionalities, and is used to create window based applications. Java swing components are lightweight, platform independent, provide powerful components like tables, scroll panels, buttons, list, color chooser, etc.

**What is JFC?**

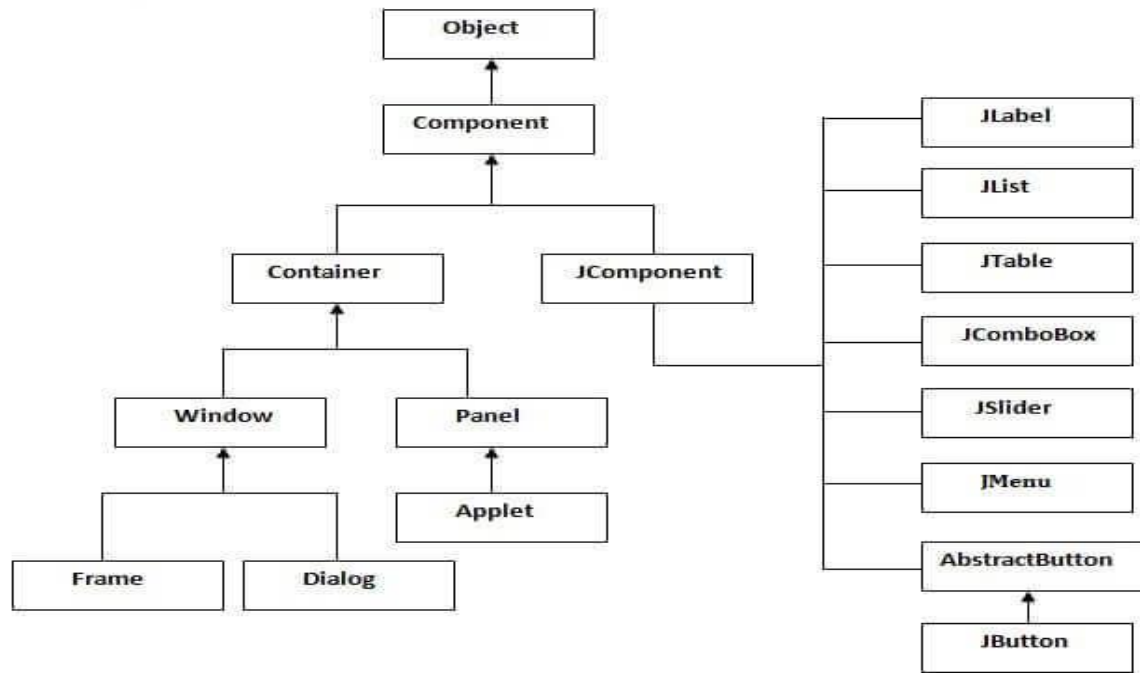
The Java Foundation Classes (JFC) are a set of GUI components which simplify the development of desktop applications.

**Difference between AWT and Swing:**

<b>Java AWT</b>	<b>Java Swing</b>
AWT components are platform-dependent.	Java swing components are platform-independent.
AWT components are heavyweight.	Swing components are lightweight.
AWT doesn't support pluggable look and feel.	Swing supports pluggable look and feel.
AWT provides less components than Swing.	Swing provides more powerful components such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
AWT doesn't follows MVC (Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing follows MVC.

## Java Swing Component Hierarchy:

The hierarchy of java swing API is given below.



### Java Swing Examples:

There are two ways to create a frame:

- 1) By creating the object of Frame class (association)
- 2) By extending Frame class (inheritance)

We can write the code of swing inside the main(), constructor or any other method.

### Simple Java Swing Example

Let's see a simple swing example where we are creating one button and adding it on the JFrame object inside the main() method.

File: FirstSwingExample.java

```

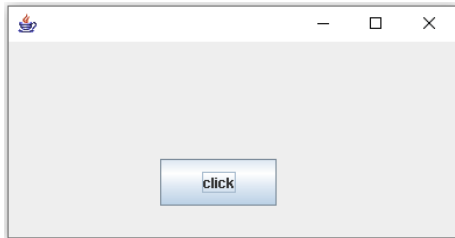
import javax.swing.*;

public class FirstSwingExample
{
    public static void main(String[] args)
    {
        JFrame f=new JFrame();           //creating instance of JFrame
        JButton b=new JButton("click");   //creating instance of JButton
        b.setBounds(130,100,100, 40);     //x axis, y axis, width, height
        f.add(b);                         //adding button in JFrame
        f.setSize(400,500);               //400 width and 500 height
    }
}
  
```

```

        f.setLayout(null);           //using no layout managers
        f.setVisible(true);         //making the frame visible
    }
}

```

**Output:****Swings components:****1) JLabel:**

The easiest control to use is a label. It is an object of type JLabel class. Labels are passive controls as it does not create any event when it is accessed. It is used to display a single line of **read only text**. The text can be changed by a programmer but a user cannot edit it directly. It inherits the **JComponent** class.

**JLabel class declaration**

```
public class JLabel extends JComponent implements SwingConstants, Accessible
```

**Constructors:**

Constructor	Description
JLabel()	It is used to create a JLabel instance with no image and with an empty string for the title.
JLabel(String s)	It is used to create a JLabel instance with the specified text..
JLabel(Icon i)	It is used to create a JLabel instance with the specified image.
JLabel(String s, Icon i, int horizontalAlignment)	It is used to create a JLabel instance with the specified text, image, and horizontal alignment.

**Commonly used Methods:**

Methods	Description
String getText()	It returns the text string that a label displays.
void setText(String text)	It defines the single line of text this component will display.
void setHorizontalAlignment(int alignment)	It sets the alignment of the label's contents along the X axis.

Icon getIcon()	It returns the graphic image that the label displays.
int getHorizontalAlignment()	It returns the alignment of the label's contents along the X axis.

**Syntax to create JLabel:**

**JLabel l = new JLabel();**

**Example**

```
import javax.swing.*;
class LabelExample
{
    public static void main(String args[])
    {
        JFrame f= new JFrame("Label Example");
        JLabel l1,l2;
        l1=new JLabel("First Label.");
        l1.setBounds(50,50, 100,30);
        l2=new JLabel("Second Label.");
        l2.setBounds(50,100, 100,30);
        f.add(l1);
        f.add(l2);
        f.setSize(300,300);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```

**Output:****2) JButton:**

This component can be used to perform some operations when the user clicks on it. When the button is pushed, the application results in some action. It basically inherits the AbstractButton class.

**JButton class declaration**

Let's see the declaration for javax.swing.JButton class.

**public class** JButton **extends** AbstractButton **implements** Accessible

**Commonly used Constructors:**

Constructor	Description
JButton()	It is used to create a button with no text and icon.
JButton(String s)	It is used to create a button with the specified text.
JButton(Icon i)	It is used to create a button with the specified icon object.

**Commonly used Methods of AbstractButton class:**

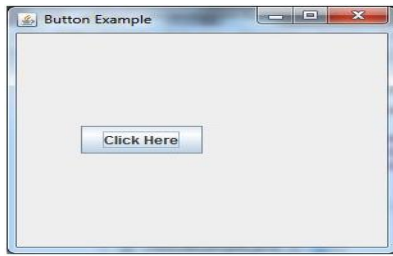
Methods	Description
void setText(String s)	It is used to set specified text on button
String getText()	It is used to return the text of the button.
void setEnabled(boolean b)	It is used to enable or disable the button.
void setIcon(Icon b)	It is used to set the specified Icon on the button.
Icon getIcon()	It is used to get the Icon of the button.

**Syntax to create JButton:**

**JButton jb = new JButton();**

**Example**

```
import javax.swing.*;
public class JButtonExample
{
    public static void main(String[] args)
    {
        JFrame f=new JFrame("Button Example");
        JButton b=new JButton("Click Here");
        b.setBounds(50,100,95,30);
        f.add(b);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```

**Output:****3) JTextField:**

The JTextField component allows the user to type some text in a single line. It basically inherits the JTextComponent class.

**JTextField class declaration**

Let's see the declaration for javax.swing.JTextField class.

**public class** JTextField **extends** JTextComponent **implements** SwingConstants

**Commonly used Constructors:**

Constructor	Description
JTextField()	Creates a new TextField
JTextField(String text)	Creates a new TextField initialized with the specified text.
JTextField(String text, int columns)	Creates a new TextField initialized with the specified text and columns.
JTextField(int columns)	Creates a new empty TextField with the specified number of columns.

**Most Commonly used method:**

Methods	Description
void setFont(Font f)	It is used to set the current font.

**Syntax to create JTextField:**

**JTextField t = new JTextField();**

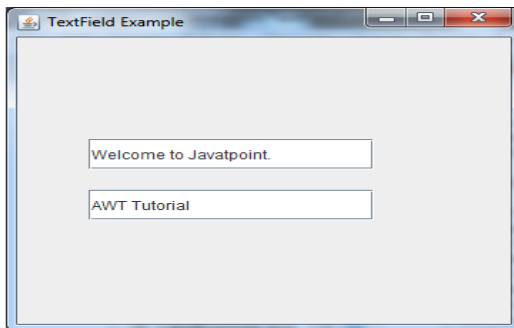
**Example**

```
import javax.swing.*;
class JTextFieldExample
{
    public static void main(String args[])
    {
        JFrame f= new JFrame("TextField Example");
        JTextField t1=new JTextField("Welcome to Javatpoint.");
    }
}
```

```

        JTextField t2=new JTextField("AWT Tutorial");
        t1.setBounds(50,100, 200,30);
        t2.setBounds(50,150, 200,30);
        f.add(t1); f.add(t2);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
}

```

**Output:****4) JTextArea:**

The JTextArea component allows the user to type the text in multiple lines. It also allows the editing of multiple-line text. It basically inherits the JTextComponent class.

**JTextArea class declaration**

Let's see the declaration for javax.swing.JTextArea class.

**public class** JTextArea **extends** JTextComponent

**Commonly used Constructors:**

Constructor	Description
JTextArea()	It is used to create a text area that displays no text initially.
JTextArea(String s)	It is used to create a text area that displays specified text initially.
JTextArea(int row, int column)	It is used to create a text area with the specified number of rows and columns that displays no text initially.
JTextArea(String s, int row, int column)	It is used to create a text area with the specified number of rows and columns that displays specified text.

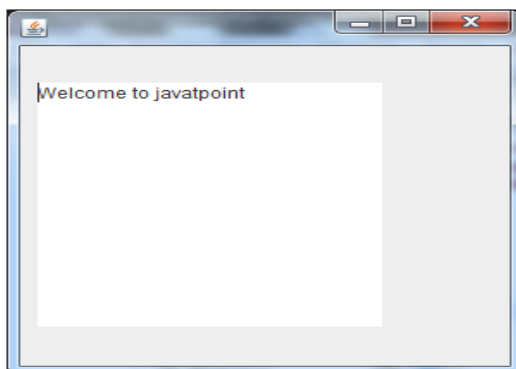


**Commonly used Methods:**

Methods	Description
void setRows(int rows)	It is used to set specified number of rows.
void setColumns(int cols)	It is used to set specified number of columns.
void setFont(Font f)	It is used to set the specified font.
void insert(String s, int position)	It is used to insert the specified text on the specified position.
void append(String s)	It is used to append the given text to the end of the document.

**Example:**

```
import javax.swing.*;
public class TextAreaExample
{
    public static void main(String args[])
    {
        JFrame f= new JFrame();
        JTextArea a=new JTextArea("Welcome to javatpoint");
        a.setBounds(10,30, 200,200);
        f.getContentPane().add(a);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //used to close swing frame
        f.setSize(300,300);
        f.getContentPane().setLayout(null);
        f.setVisible(true);
    }
}
```

**Output:**

## Layout Managers:

The Layout Managers are used to arrange components in a particular manner. The **Layout Managers** provide the facility to arrange the position and size of the components in GUI forms. For example, Setlayout is a method used to set the layout of the containers. Java provides the various layouts.

Some important layouts:

- 1) Flow Layout
- 2) Border Layout
- 3) Grid Layout
- 4) Box Layout
- 5) Card Layout
- 6) Grid Bag Layout
- 7) Group Layout

- 1) **FlowLayout:** The FlowLayout is used to arrange the components in a line, one after another. It is the default layout of applet or panel.

### Fields

1. public static final int LEFT
2. public static final int RIGHT
3. public static final int CENTER
4. public static final int LEADING
5. public static final int TRAILING

### Constructors:

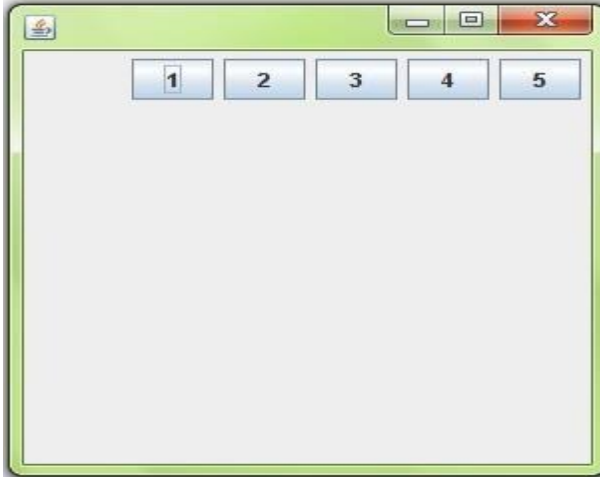
- i) **FlowLayout()** - creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap.
- ii) **FlowLayout(int align)** - creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.
- iii) **FlowLayout(int align, int hgap, int vgap)** - creates a flow layout with the given alignment and the given horizontal and vertical gap.

### Example:

```
import java.awt.*;  
import javax.swing.*;  
public class MyFlowLayout
```

```
{  
    public static void main(String[] args)  
    {  
        JFrame f=new JFrame();  
        JButton b1=new JButton("1");  
        JButton b2=new JButton("2");  
        JButton b3=new JButton("3");  
        JButton b4=new JButton("4");  
        JButton b5=new JButton("5");  
        f.add(b1);  
        f.add(b2);  
        f.add(b3);  
        f.add(b4);  
        f.add(b5);  
        f.setLayout(new FlowLayout(FlowLayout.LEADING));  
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        f.setSize(300,300);  
        f.setVisible(true);  
    }  
}
```

**Output:**



## 2) BorderLayout:

The BorderLayout is used to arrange the components in five regions: north, south, east, west, and center. Each region (area) may contain one component only. It is the default layout of a frame or window. The BorderLayout provides five constants for each region:

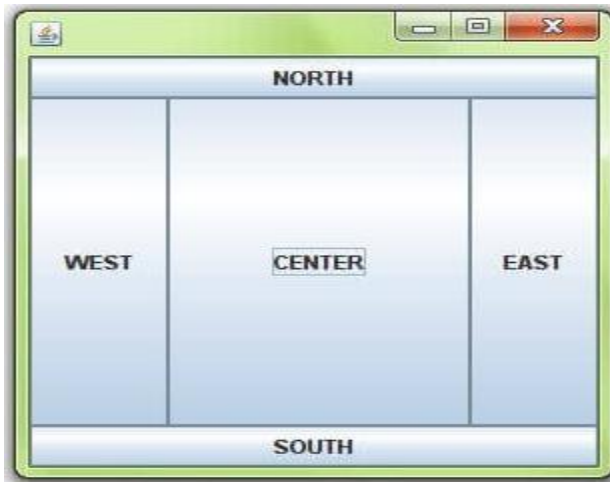
- public static final int NORTH
- public static final int SOUTH

- public static final int EAST
- public static final int WEST
- public static final int CENTER

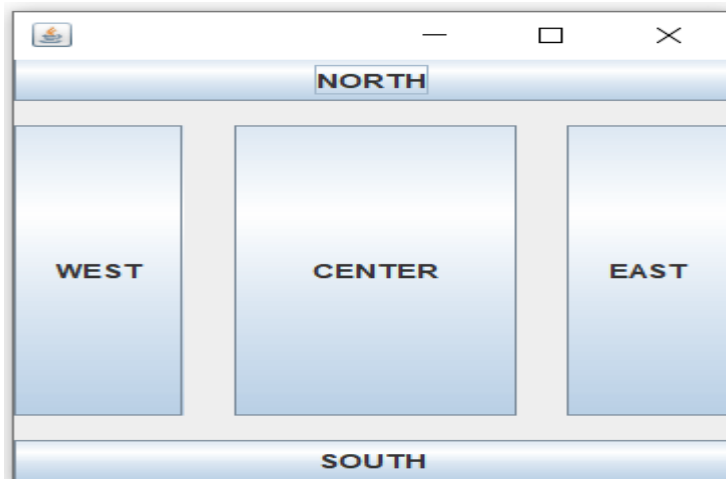
**Constructors of BorderLayout class:**

- **BorderLayout():** creates a border layout but with no gaps between the components.
- **BorderLayout(int hgap, int vgap):** creates a border layout with the given horizontal and vertical gaps between the components.

```
import java.awt.*;
import javax.swing.*;
public class MyFlowLayout
{
    public static void main(String[] args)
    {
        JFrame f=new JFrame();
        JButton b1 = new JButton("NORTH");// the button will be labeled as NORTH
        JButton b2 = new JButton("SOUTH");// the button will be labeled as SOUTH
        JButton b3 = new JButton("EAST");// the button will be labeled as EAST
        JButton b4 = new JButton("WEST");// the button will be labeled as WEST
        JButton b5 = new JButton("CENTER");// the button will be labeled as CENTER
        //f.setLayout(new BorderLayout(20, 15));
        f.add(b1, BorderLayout.NORTH);// b1 will be placed in the North Direction
        f.add(b2, BorderLayout.SOUTH);// b2 will be placed in the South Direction
        f.add(b3, BorderLayout.EAST);// b3 will be placed in the East Direction
        f.add(b4, BorderLayout.WEST);// b4 will be placed in the West Direction
        f.add(b5, BorderLayout.CENTER);// b5 will be placed in the Center
        f.setSize(300, 300);
        f.setVisible(true);
    }
}
```

**Output:**

If you set `f.setLayout(new BorderLayout(20, 15))` as layout then output will be as like follow

**3) GridLayout:**

The Java GridLayout class is used to arrange the components in a rectangular grid. One component is displayed in each rectangle. GridLayout divides the container into rows and columns. The intersection of the row and the column is called a cell. The cell can contain only one component. The position of the component in a grid is determined by the order in which components are added to the grid.

**Constructors of GridLayout Class**

- **GridLayout()** - creates a grid layout with one column per component in a row.
- **GridLayout(int rows, int columns)** - creates a grid layout with the given rows and columns but no gaps between the components.
- **GridLayout(int rows, int columns, int hgap, int vgap)** - creates a grid layout with the given rows and columns along with given horizontal and vertical gaps.

**Example**

```
import java.awt.*;
import javax.swing.*;
public class MyFlowLayout
{
    public static void main(String[] args)
    {
        JFrame f=new JFrame();
        JButton b1 = new JButton("1");
        JButton b2 = new JButton("2");
        JButton b3 = new JButton("3");
        JButton b4 = new JButton("4");
        f.add(b1);
        f.add(b2);
        f.add(b3);
        f.add(b4);
        f.setLayout(new GridLayout());
        f.setSize(300, 300);
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

**Output:****4) BorderLayout:**

The **Java BorderLayout** class is used to arrange the components either vertically or horizontally. For this purpose, the BorderLayout class provides four constants. They are as follows:

**Fields of BorderLayout Class**

- **public static final int X\_AXIS:** Alignment of the components are horizontal from left to right.
- **public static final int Y\_AXIS:** Alignment of the components are vertical from top to bottom.

#### Constructor of BorderLayout class:

- **BoxLayout(Container obj, int axis):** creates a box layout that arranges the components with the given axis.

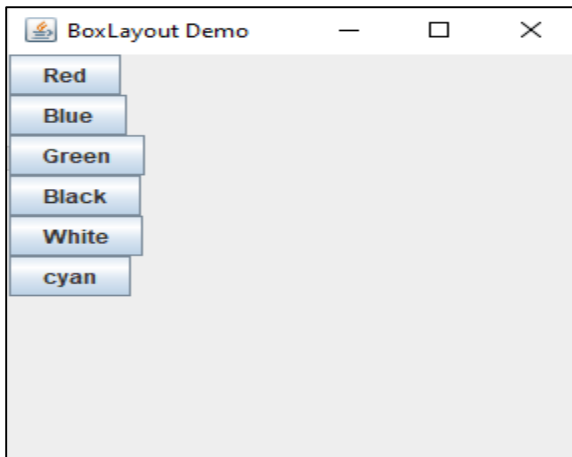
Where axis specifies the direction, for horizontal direction you can use

**BoxLayout.X\_AXIS** and for vertical direction you can use **BoxLayout.Y\_AXIS**

#### Example

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

public class BorderLayoutDemo extends JFrame
{
    public static void main(String args[])
    {
        JButton red, blue, green, black, white, cyan;
        JFrame f=new JFrame();
        red = new JButton("Red");
        blue = new JButton("Blue");
        green = new JButton("Green");
        black = new JButton("Black");
        white = new JButton("White");
        cyan = new JButton("cyan");
        f.add(red);
        f.add(blue);
        f.add(green);
        f.add(black);
        f.add(white);
        f.add(cyan);
        f.setLayout(new BorderLayout(f.getContentPane(), BorderLayout.Y_AXIS));
        f.setVisible(true);
        f.setSize(300, 300);
        f.setTitle("BoxLayout Demo");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

**Output:****5) CardLayout:**

A card layout represents a stack of cards displayed on a container. At a time only one card can be visible and each can contain the only one component.

**Creation of Card Layout in Java:**

```
CardLayout cl = new CardLayout();
```

```
CardLayout cl = new CardLayout(int hgap, int vgap);
```

**To add the components in CardLayout we use add method:**

```
add("Cardname", Component);
```

**Methods of CardLayout in Java:**

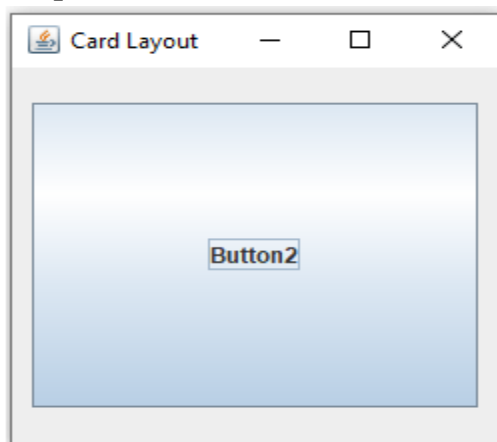
- 1) **first(Container):** It is used to flip to the first card of the given container.
- 2) **last(Container):** It is used to flip to the last card of the given container.
- 3) **next(Container):** It is used to flip to the next card of the given container.
- 4) **previous(Container):** It is used to flip to the previous card of the given container.
- 5) **show(Container, cardname):** It is used to flip to the specified card with the given name.

**Example:**

```
import java.awt.*;  
import javax.swing.*;  
import java.awt.event.*;  
public class CardLayoutDemo extends JFrame implements ActionListener  
{  
    JButton b1, b2, b3, b4, b5;  
    CardLayout cl;  
    Container c;  
    CardLayoutDemo ()  
    {
```



```
b1 = new JButton ("Button1");
b2 = new JButton ("Button2");
b3 = new JButton ("Button3");
b4 = new JButton ("Button4");
b5 = new JButton ("Button5");
f=new JFrame("card Layout");
cl=new CardLayout (10, 20);
f.setLayout(cl);
f.add ("Card1", b1);
f.add ("Card2", b2);
f.add ("Card3", b3);
b1.addActionListener (this);
b2.addActionListener (this);
b3.addActionListener (this);
f.setVisible (true);
f.setSize (400, 400);
f.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);  }
public void actionPerformed (ActionEvent ae)
{
    cl.next (f.getContentPane());
}
public static void main (String[]args)
{
    CardLayoutDemo x=new CardLayoutDemo();
}
}
```

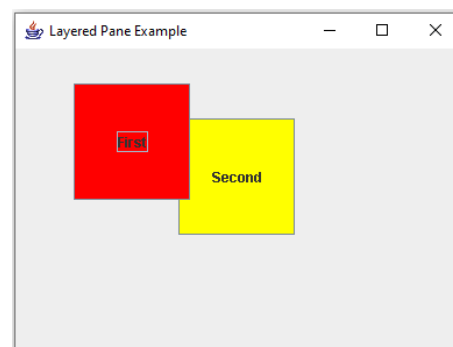
**Output:**

## Panes in Java:

A layered pane is a Swing container which is used to hold the various components using the concept of layers. The components present in the upper layer overlaps the components present in the lower layer. The layered pane is created using the `JLayeredPane` class the only constructor of this class is `JLayeredPane ()`.

```
import java.awt.*;
import javax.swing.*;
class PaneExample extends JApplet
{
    JFrame jf ;
    JLayeredPane LPane;
    JButton first, second;
    PaneExample()
    {
        jf =new JFrame("Layered Pane Example");
        LPane =new JLayeredPane();
        jf.getContentPane() .add(LPane);
        first= new JButton("First");
        first.setBackground(Color.red);
        first.setBounds(50,30,100,100);
        second= new JButton("Second");
        second.setBackground(Color.yellow);
        second.setBounds(140,60,100,100);
        LPane.add(first, new Integer(3));
        LPane.add(second, new Integer(2));
        jf.setDefaultCloseOperation(jf.EXIT_ON_CLOSE);
        jf.setSize (400,300) ;
        jf.setVisible(true);
    }
    public static void main(String args[])
    {
        PaneExample le= new PaneExample();
    }
}
```

Output:



\*\*\*\*\*