UNIT-1

OOP Principles:

The following are the basic oops concepts:
1. Class
2.Object
3.Data Abstraction.
4.Data Encapsulation.
5.Inheritance.
6.Polymorphism.

1.Class

Class is a blue print which is containing only list of variables and methods and no memory is allocated for them. A class is a group of objects that has common properties.

2.Object

Any entity that has state and behavior is known as an object. For example a chair, pen, table, keyboard, bike, etc. It can be physical or logical. An Object can be defined as an instance of a class.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

3.Data Abstraction

Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing.
In Java, we use abstract class and interface to achieve abstraction.

4.Data Encapsulation

Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example capsule, it is wrapped with different medicines.
A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

5.Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Benefits of Inheritance

- Software Reusability (among projects)
- Increased Reliability (resulting from reuse and sharing of well-tested code)
- Code Sharing (within a project)
- Consistency of Interface (among related objects)
- Software Components
- Rapid Prototyping (quickly assemble from pre-existing components)

- Polymorphism and Frameworks (high-level reusable components)
- Information Hiding

6.Polymorphism

If *one task is performed by different ways,* it is known as polymorphism.
For example:
To convince the customer differently,
To draw something, for example, shape, triangle, rectangle, etc.
In Java, we use method overloading and method overriding to achieve polymorphism.

Procedural Vs Object Oriented Programming

Procedure Oriented Programming	**Object Oriented Programming**
1. In POP, program is divided into small parts called functions.	1. In OOP, program is divided into parts called objects.
2. In POP,Importance is not given to data but to functions as well as sequence of actions to be done.	2. In OOP, Importance is given to the data rather than procedures or functions because it works as a real world.
3. POP follows Top Down approach.	3. OOP follows Bottom Up approach.
4. POP does not have any access specifier.	4. OOP has access specifiers named Public, Private, Protected, etc.
5. In POP, Data can move freely from function to function in the system.	5. In OOP, objects can move and communicate with each other through member functions.
6. To add new data and function in POP is not so easy.	6. OOP provides an easy way to add new data and function.
7. In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	7. In OOP, data can not move easily from function to function,it can be kept public or private so we can control the access of data.
8. POP does not have any proper way for hiding data so it is less secure.	8. OOP provides Data Hiding so provides more security.
9. In POP, Overloading is not possible.	9. In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
10. Example of POP are : C, VB, FORTRAN, Pascal.	10. Example of OOP are : C++, JAVA, VB.NET, C#.NET.

Java Buzzwords

Following are the list of buzzwords:

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Simple:

Java was designed to be easy for the professional programmer to learn and use effectively.
Learning Java will be even easier, if we understand the concepts of Object oriented programming because Java inherits the C/C++ syntax and many of the object-oriented features of C++.

Secure

It is a more secure language compared to other language; In this language, all code is covered in byte code after compilation which is not readable by human.

Portability

If any language supports platform independent and architectural neutral feature known as portable. The languages like C, CPP, Pascal are treated as non-portable language. It is a portable language.

Object-Oriented

Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.

Robust

Simply means of Robust are strong. It is robust or strong Programming Language because of its capability to handle Run-time Error, automatic garbage collection, the lack of pointer concept, Exception Handling. All these points make It robust Language.

Multithreaded

Java supports multithreaded programming, which allows you to write programs that do many things simultaneously.
A multithreaded program contains two or more parts that can run concurrently. Each part of such program is called a thread.

Architecture-Neutral

Architecture represents processor. A Language or Technology is said to be Architectural neutral which can run on any available processors in the real world without considering their development and compilation.

Interpreted and High Performance

It have high performance because of following reasons;

- This language **uses Bytecode** which is faster than ordinary pointer code so Performance of this language is high.
- **Garbage collector**, collect the unused memory space and improve the performance of the application.
- It has **no pointers** so that using this language we can develop an application very easily.
- It **support multithreading**, because of this time consuming process can be reduced to executing the program.

Distributed

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.

Dynamic

It supports Dynamic memory allocation, due to this memory wastage is reduce and improve performance of the application. The process of allocating the memory space to the input of the program at a run-time is known as dynamic memory allocation, To allocate memory space by dynamically we use an operator called 'new' 'new' operator is known as dynamic memory allocation operator.

History Of Java

Java was developed by James Gosling, Patrick Naughton, Chris warth, Ed Frank and Mike Sheridon at Sun Microsystems in the year 1991.

This language was initially called as “OAK” but was renamed as “Java” in 1995.Between this initial design of oak and the public announcement of java i.e. in between 1991 and 1995, many more developers contributed to the design and evolution of java.

The primary motivation behind developing java was the need for creating a platform independent Language (Architecture Neutral), that can be used to create a software which can be embedded in various electronic devices such as remote controls, micro ovens etc.

The problem with C, C++ and most other languages is that, they are designed to compile on specific targeted CPU (i.e. they are platform dependent), but java is platform Independent which can run on a variety of CPU’s under different environments.

The secondary factor that motivated the development of java is to develop the applications that can run on Internet. Using java we can develop the applications which can run on internet i.e. Applet. So java is a platform Independent Language used for developing programs which are platform Independent and can run on internet.

JVM

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS

is different from each other. However, Java is platform independent. There are three notions of the JVM: *specification*, *implementation*, and *instance*.

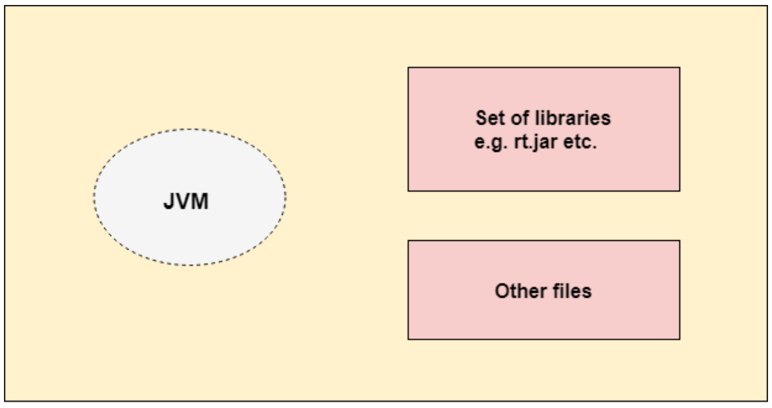
The JVM performs the following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

The implementation of JVM is also actively released by other companies besides Sun Micro Systems.

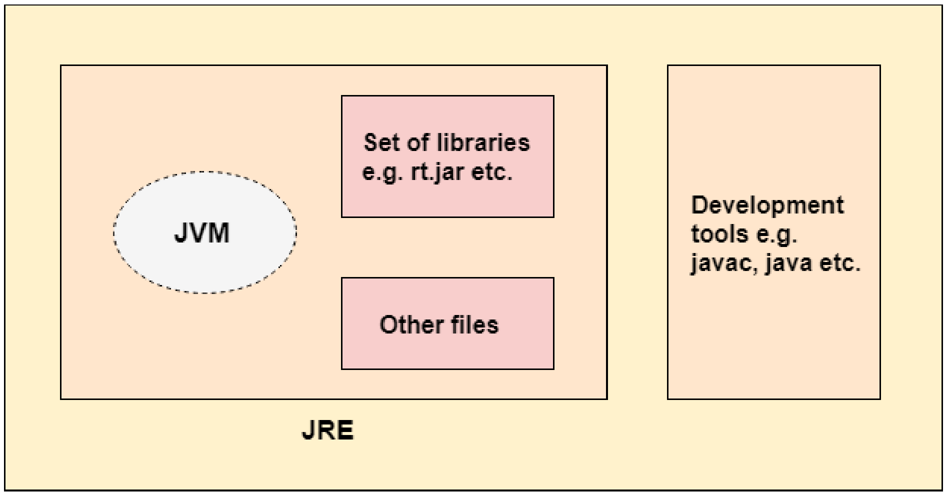


JRE

JDK

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools.

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.


Set of libraries e.g. rt.jar etc.
JVM
Other files
Development tools e.g. javac, java etc.
JRE

JDK

JVM (Java Virtual Machine) Architecture:

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

JVM is :

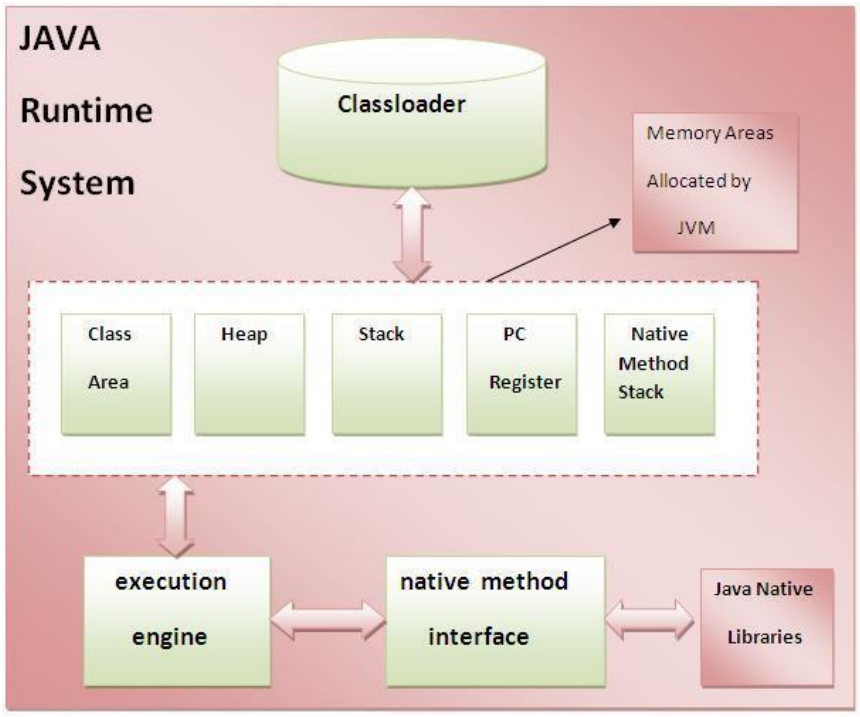
1. A specification where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.
2. An implementation Its implementation is known as JRE (Java Runtime Environment).
3. Runtime Instance Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

The JVM performs following operation:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.



1) Class loader
 Class loader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the class loader.

2) Class(Method) Area
Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

3) Heap
It is the runtime data area in which objects are allocated.

4) Stack
Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.

5) Program Counter Register
PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

6) Native Method Stack
It contains all the native methods used in the application.

7) Execution Engine
It contains:

1. A virtual processor
2. Interpreter: Read bytecode stream then execute the instructions.
3. Just-In-Time(JIT) compiler: It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.
4. Garbage Collector: Automatic freeing of Heap Memory

8) Java Native Interface
Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc. Java uses JNI framework to send output to the Console or interact with OS libraries.

Comments

There are 3 types of comments in java.

1.Single Line Comment
2.Multi Line Comment
3.Documentation Comment

1.Java Single Line Comment

The single line comment is used to comment only one line.

Syntax:

//This is single line comment

Example:

```
public class CommentExample1
 {
  public static void main(String[] args)
   {
      int i=10;//Here, i is a variable
     System.out.println(i);
   }
 }
```

2.Java Multi Line Comment

The multi line comment is used to comment multiple lines of code.

Syntax:

```
/*
This
is
multi line
comment
*/
```

Example:

```
public class CommentExample2
{
public static void main(String[] args)
{
/* Let's declare and
print variable in java. */
int i=10;
System.out.println(i);
}
}
```

3. Java Documentation Comment

The documentation comment is used to create documentation API. To create documentation API, you need to use **javadoc tool**.

Syntax:

```
/**
This
is
documentation
comment
*/
```

Datatypes

Java defines eight primitive types of data: byte, short, int, long, char, float, double, and Boolean. The primitive types are also commonly referred to as simple types.

Integers : This group includes byte, short, int, and long.

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

Syntax: byte b, c;

```
        int i;
        short s;
        long distance;
```

Example:

```
class Light
{
public static void main(String args[])
{
int lightspeed = 186000;
long days= 1000, seconds, distance;
// approximate speed of light in miles per second
seconds = days * 24 * 60 * 60;
distance = lightspeed * seconds;
System.out.print("In " + days);
System.out.print(" days light will travel about "+ distance + " miles.”);
} }
```

output: In 1000 days light will travel about 16070400000000 miles.

Floating point types: Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision.

There are two kinds: float and double.

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

Syntax:
```
float hightemp, lowtemp;
Double radius, pi;
```
Example: // Compute the area of a circle.
```
class Area {
public static void main(String args[]) {
double pi=3.1416, r  = 10.8, a;
a = pi * r * r;
System.out.println("Area of circle is " + a); }        }
```

Characters: In Java, the data type used to store characters is char.
Width of char: 16 bit.
Range of char: 0 to 65,536.
There are no negative chars in java.
Syntax: car c=’A’, c1=’X’, c2=’Z’;

Boolean: Java has a primitive type, called Boolean, for logical values. It can have only one of two possible values, true or false.

Example: // Demonstrate boolean values.

```
class BoolTest {
public static void main(String args[]) {
boolean b;
b = false;
System.out.println("b is " + b);
b = true;
System.out.println("b is " + b);
if(b)
 System.out.println("This is executed.");   } }
```

Variables

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer.

Declaring a Variable:

In Java, all variables must be declared before they can be used.

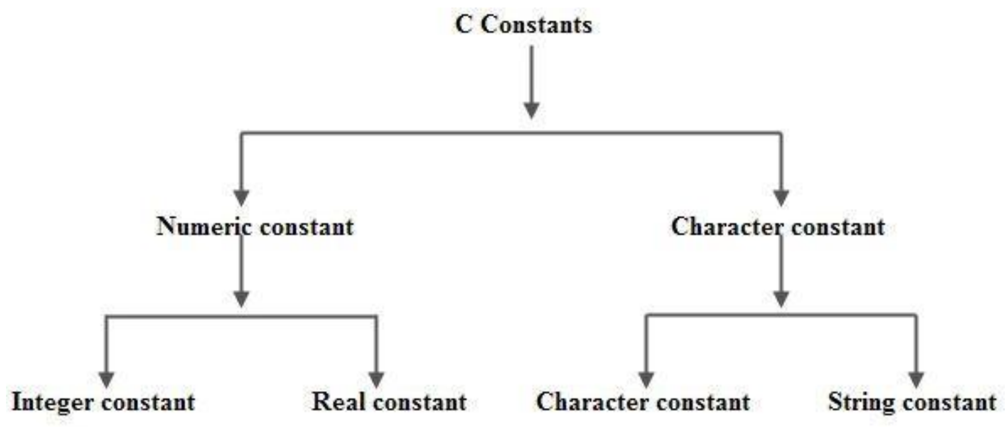
Syntax: type identifier [= value][, identifier [= value] ...] ;

The type is one of Java's atomic types, or the name of a class or interface
The identifier is the name of the variable.

Examples of variable declarations of various types.

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

Constants



Integer Constants:

An integer constant is a sequence of digits from 0 to 9 without decimal points or fractional part or any other symbols.
There are 3 types of integers namely decimal integer, octal integers and hexadecimal integer.

Decimal Integers consists of a set of digits 0 to 9 preceded by an optional + or - sign. Spaces, commas and non digit characters are not permitted between digits.
Example int y=123; //here 123 is a decimal integer constant

Octal Integers constant consists of any combination of digits from 0 through 7 with a O at the beginning.
Example
int X=O123; // here 0123 is a octal integer constant .

Hexadecimal integer constant is preceded by OX or Ox, they may contain alphabets from A to F or a to f.
The alphabets A to F refers to 10 to 15 in decimal digits.
Example
int x=Ox12 // here Ox12 is a Hexa-Decimal integer constant

Real Constants:

Real Constants consists of a fractional part in their representation. These quantities are represented by numbers containing fractional parts like 26.082.

Example
float x = 6.3; //here 6.3 is a double constant.
float y = 6.3f; //here 6.3f is a float constant.

Single Character Constants:

A Single Character constant represent a single character which is enclosed in a pair of quotation symbols.

```
char p ='ok' ;  // p will hold the value 'O' and k will be omitted
char y ='u';    // y will hold the value 'u'
char k ='34' ; // k will hold the value '3, and '4' will be omitted
char e =' ';     // e will hold the value ' ' , a blank space
chars ='\45'; // swill hold the value ' ' , a blank space
```

All character constants have an equivalent integer value which are called ASCII Values.

String Constants

A string constant is a set of characters enclosed in double quotation marks. The characters in a string constant sequence may be a alphabet, number, special character and blank space. Example of string constants are

"VISHAL"   "1234" "God Bless" "!.....?"

Scope and Lifetime of Variables

The scope of a variable defines the section of the code in which the variable is visible. As a general rule, variables that are defined within a block are not accessible outside that block.

There are three types of variables:

1) instance variables

2) class variables

3) local variables

1.Instance Variables

A variable which is declared inside a class and outside all the methods and blocks is an instance variable.

General **scope of an instance variable** is throughout the class except in static methods. **Lifetime of an instance variable** is until the object stays in memory.

2.Class Variables

A variable which is declared inside a class, outside all the blocks and is marked *static* is known as a class variable.

General scope of a class variable is throughout the class and the **lifetime** of a class variable is until the end of the program or as long as the class is loaded in memory.

The lifetime of a variable refers to how long the variable exists before it is destroyed. Destroying variables refers to deallocating the memory that was allotted to the variables when declaring it.

3.Local Variables

All other variables which are not instance and class variables are treated as local variables including the parameters in a method.

Scope of a local variable is within the block in which it is declared and the **lifetime of a local variable** is until the control leaves the block in which it is declared.

Operators

Operator in Java is a symbol that is used to perform operations. For example: +, -, *, / etc.

here are many types of operators in Java which are given below:

- Unary Operator(Increment and decrement),
- Arithmetic Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator
- Assignment Operator.

1.Arithmetic Operators: The following table lists the arithmetic operators:

Operator	**Result**
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

Example:// Demonstrate the basic arithmetic operators.

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

```
System.out.println("Integer Arithmetic");
int a = 1 + 1;
int b = a * 3;
int c = b / 4;
int d = c - a;
int e = -d;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
System.out.println("e = " + e);  }}
```

2.Bitwise Operators:

Operator	**Result**
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

3.Relational Operators

Operator	**Result**
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

4.Boolean Logical Operators

Operator	**Result**

&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

Arrays:

An Array is the collection of elements of similar datatype which are referred by a common name.

Arrays of any type can be created and may have one or more dimensions. An element in an array can be accessed by using its index.

1.One Dimensional Array:

A one dimensional array is essentially a list of like typed variables.

Syntax: datatype arrayname[];

Here, type declares the base type of the array. The base type determines the data type of each element that comprises the array.
Array_name refers to name of array, and all elements are referred by this common name.

Eg: int monthdays[];
Although in the above example it declares an array variable with name monthdays, but no array actually exists. In fact, the value of month_days is set to null, which represents an array with no value.

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

Syntax: arrayname=new datatype(size)
Monthdays=new int[12];

Now monthdays will refer to an array of 12 integers.

Example Program:
class Array{

```
public static void main(String args[]) {
int month_days[];
month_days = new int[12];
month_days[0] = 31;
month_days[1] = 28;
month_days[2] = 31;
month_days[3] = 30;
month_days[4] = 31;
month_days[5] = 30;
month_days[6] = 31;
month_days[7] = 31;
month_days[8] = 30;
month_days[9] = 31;
month_days[10] = 30;
month_days[11] = 31;
System.out.println("April has " + month_days[3] + " days.");  }}
```

Arrays can be initialized at the time of declaration.
An Array Initializer is a list of comma-separated expressions surrounded by curly braces.

Example:

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

2.Multidimensional Arrays

In Java, multidimensional arrays are actually arrays of arrays.

Syntax : twoD[][]=new int[4][5];
This creates a 4 by 5 array and assigns to twoD array.

Example:

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

Control Statements:

Control statements can be put into the following categories:

1.Selection Statements
2. Iteration Statements
3. Jump Statements

1.Selection statements:

Java supports two selection statements: if and switch.

if : The if statement is conditional branch statement. It can be used to route program execution through two different paths.

Syntax:

```
if(condition)
  statement1;
else
  statement2;
```

Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a boolean value. The else
clause is optional.

The if works like this: If the condition is true, then statement1 is executed. Otherwise,statement2 (if it exists) is executed. In no case will both statements be executed.

Example:

```
int a, b;
if(a < b)
   a = 0;
else
    b = 0;
```

Here, if a is less than b, then a is set to zero. Otherwise, b is set to zero.

Nested ifs: A nested if is an if statement that is the target of another if or else. Nested ifs are very commonin programming. When you nest ifs an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else.

Eg: `if(i == 10) {`

```
    if(j < 20) a = b;
    if(k > 100) c = d;
    else a = c;
    }
   else
    a = d;
```

if-else-if ladder:

Syntax:
```
if(condition)
statement;
else if(condition)
statement;
else if(condition)
statement;
…..
else
statement;
```

The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is by passed.

Switch

The switch statement isa multiway branch statement.

Syntax:
```
switch (expression) {
case value1:
        // statement sequence
        break;
case value2:
        // statement sequence
        break;
 ….
case valueN:
```

The value of the expression is compared with each of the literal values in the case statements. If a match is found, the code sequence following that case statement is executed. If none of the constants matches the value of the expression, then the default statement is executed. However, the default statement is optional. If no case matches and no default is present, then no further action is taken.

Example:

```
class SampleSwitch
{
public static void main(String args[]) {
for(int i=0; i<6; i++)
switch(i)
{
case 0:
System.out.println("i is zero.");
break;
case 1:
System.out.println("i is one.");
break;
case 2:
System.out.println("i is two.");
default:
System.out.println("i is greater than 3.");
}
}  }
```

2.Iteration Statements:

Java's iteration statements are for, while, and do-while.

While:

It repeats a statement or block while its controlling expression is true.

Syntax:

```
while(condition)
  {
        // body of loop
   }
```

The condition can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop.

Eg:

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

do-while:

The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop. If this expression is true, the loop will repeat. Otherwise, the loop terminates.

Syntax:
```
                    do
                         {
                      // body of loop
                          } while (condition);
```
Eg :
```
class DoWhile {
public static void main(String args[]) {
int n = 10;
do {
System.out.println("tick " + n);
n--;
} while(n > 0);
}}
```

for:

Syntax:
```
for(initialization; condition; inc/dec)
 {
      // body
 }
```
If only one statement is being repeated, there is no need for the curly braces.

The for loop operates as follows. When the loop first starts, the initialization portion of the loop is executed. It is important to understand that the initialization expression is only executed once. Next,condition is evaluated. If this condition is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

Eg:
```
class ForTick {
public static void main(String args[]) {
int n;
for(n=10; n>0; n--)
System.out.println("tick " + n);
}  }
```
Using comma in for loop:
```
class Comma {
public static void main(String args[]) {
int a, b;
for(a=1, b=4; a<b; a++, b--)
 {
```

```
System.out.println("a = " + a);
System.out.println("b = " + b);
}  }
}
```

3.Jump Statements:

break: When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

Example:

```
class BreakStatement
{
 public static void main(String args[] )
        {
                int i;
                i=1;
                while(true)
                {
                        if(i >10)
                        break;
                        System.out.print(i+" ");
                        i++;
                }
        }
}
```

Continue:

This command skips the whole body of the loop and executes the loop with the next iteration. On finding continue command, control leaves the rest of the statements in the loop and goes back to the top of the loop to execute it with the next iteration (value).

Example:

```
/* Print Number from 1 to 10 Except 5 */
class NumberExcept
{
        public static void main(String args[] )
        {
                int i;
                for(i=1;i<=10;i++)
                {
                        if(i==5) continue;
                        System.out.print(i +" ");
                }
        }
}
```

Type conversion and Casting:

Type Casting is required whenever we assigning smaller data type value into bigger data type variable or assigning bigger data type value to the smaller data type variable. There are two types of Type Casting they are;

1.Implicit Type Casting

2.Explicit Type Casting

1.Implicit Type Casting

This Type Casting is required whenever we assigning smaller data type value into bigger data type variable. It is also known as **widening or upcasting**

Note: In this type casting no loose of information.

2.Explicit Type Casting

Programmer is responsible to perform this type casting.

This Type Casting is required whenever we assigning bigger data type value to the smaller data type variable. It is also known as **narrowing or down casting**

Syntax: (target-type) value

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

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

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

Concepts of classes, objects

Here is a class called Box that defines three instance variables: width, height, and depth. Currently, Box does not contain any methods.

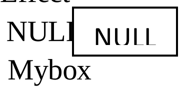
```
class Box
 {
double width;
double height;
double depth;
}
```

Declaring object: To actually create a Box object, you will use a statement like the following:

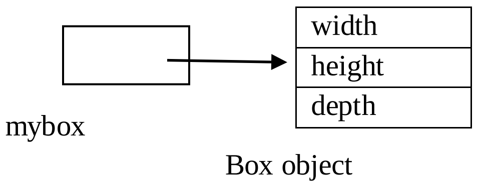
```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

Statement Effect

Box mybox;



Mybox=new Box();



Box mybox = new Box(); // create a Box object called mybox
the new operator dynamically allocates memory for an object.

Example:

```
class Box
{
double width;
double height;
double depth;
}
class BoxDemo
 {
public static void main(String args[])
{
Box mybox = new Box();
double vol;
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;
vol = mybox.width * mybox.height * mybox.depth;
System.out.println("Volume is " + vol);
}
}
```

Methods: classes usually consist of two things: instance variables and methods.

Syntax:
```
returntype methodname(parameter-list)
 {

       // body of method

 }
```
Here, type specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be void.

Methods that have a return type other than void return a value to the calling routine using the following form: return value;

Example:
```
//Adding a Method to the Box Class
Class Box
{
double width, height, depth;
void volume() {
System.out.print("Volume is ");
System.out.println(width * height * depth);
}    }
class BoxDemo3 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
mybox1.volume();
mybox2.volume();
}    }
```

Example: Program for Returning a Value and adding a method that takes parameters

```
class Box
{
double width, height, depth;
double volume() {
return width * height * depth;
}
void setDim(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
}
class BoxDemo5 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
mybox1.setDim(10, 20, 15);
mybox2.setDim(3, 6, 9);
vol = mybox1.volume();
System.out.println("Volume is " + vol);
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}  }
```

Method Overloading:

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading.

Method overloading is one of the ways that Java supports polymorphism.

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.

Example:

```
class OverloadDemo
```

```
 {
    void test() {
    System.out.println("No parameters");
    }
void test(int a) {
System.out.println("a: " + a);
}
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
double test(double a) {
System.out.println("double a: " + a);
return a*a;
}  }
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}
}
```

Constructors

constructor in Java is a special member method which will be called implicitly (automatically) by the JVM whenever an object is created for placing user or programmer defined values in place of default values. In a single word constructor is a special member method which will be called automatically whenever object is created.

The purpose of constructor is to initialize an object called object initialization. Constructors are mainly create for initializing the object. Initialization is a process of assigning user defined values at the time of allocation of memory space.

Syntax

```
className()
{
.......
.......
}
```

Difference between Method and Constructor

	Method	Constructor
1	Method can be any user defined name	Constructor must be class name
2	Method should have return type	It should not have any return type (even void)
3	Method should be called explicitly either with object reference or class reference	It will be called automatically whenever object is created
4	Method is not provided by compiler in any case.	The java compiler provides a default constructor if we do not have any constructor.

Types of constructors

Based on creating objects in Java constructor are classified in two types. They are

1.Default or no argument Constructor

2.Parameterized constructor.

1.Default Constructor

A constructor is said to be default constructor if and only if it never take any parameters.

If any class does not contain at least one user defined constructor than the system will create a default constructor at the time of compilation it is known as system defined default constructor.

Note: System defined default constructor is created by java compiler and does not have any statement in the body part. This constructor will be executed every time whenever an object is created if that class does not contain any user defined constructor.

Example:

```
class Test
{
int a, b;
Test ()
{
a=10;
b=20;
System.out.println("Value of a: "+a);
System.out.println("Value of b: "+b);
}
};
class TestDemo
{
public static void main(String [] args)
{
Test t1=new Test ();
}
};
```

2.Parameterized constructor

If any constructor contain list of variable in its signature is known as paremetrized constructor. A parameterized constructor is one which takes some parameters.

Example:

```
class Test
{
int a, b;
Test(int n1, int n2)
{
a=n1;
b=n2;
System.out.println("Value of a = "+a);
System.out.println("Value of b = "+b);
}
};
class TestDemo1
{
public static void main(String k [])
{
Test t1=new Test(10, 20);
}
};
```

Constructor Overloading:

In addition to overloading normal methods, you can also overload constructor methods.
We can define multiple constructors with same class name but they should differ by either type or number of parameters.

Example:

```
class Box
{
    double width, height, depth;
    Box(double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }
    Box()
    {
        width = height = depth = 0;
    }
    Box(double len)
    {
        width = height = depth = len;
    }
    double volume()
    {
        return width * height * depth;
    }
}
public class Test
{
    public static void main(String args[])
    {
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mybox3 = new Box(7);
        double vol;
        vol = mybox1.volume();
        System.out.println(" Volume of mybox1 is " + vol);
        vol = mybox2.volume();
        System.out.println(" Volume of mybox2 is " + vol);
        vol = mybox3.volume();
        System.out.println(" Volume of mybox3 is " + vol);
    }
}
```

“this” keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the “this” keyword.

That is, this is always a reference to the object on which the method was invoked.

Keyword “this” is used to solve the problem of Instance variable hiding.

Instance Variable Hiding
It is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.when a local variable has the same name as an instance variable, the local variable hides the instance variable.
While it is usually easier to simply use different names, there is another way to solve the situation i.e. by using “this” you can resolve any name space collisions that might occur between instance variables and local variables.

Example:

```
Class Box
{
double  width, height, depth;
Box(double width, double height, double depth) {
this.width = width;
this.height = height;
this.depth = depth;
}
double volume()
{
return width*height*depth;
}
}
Class DemoOnthis
{
public static void main(String args[])
{
  Box ob=new Box();
  System.out.println(“volume of a box=”+ob.volume());
  }  }
```

Garbage Collection:

Objects in java are dynamically allocated by using the new operator, but how such objects are destroyed and their memory released for later reallocation.

In some languages, such as C++, dynamically allocated objects must be manually deleted by use of a delete operator. Java takes a different approach; it handles deallocation automatically. The technique that accomplishes this is called garbage collection.

It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.

The finalize( ) Method:
Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called finalization. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

Inside the finalize( ) method, you will specify those actions that must be performed before an object is destroyed.

Syntax:
```
protected void finalize( )
{
    // finalization code here
 }
```
Here, the keyword protected is a specifier that prevents access to finalize( ) by code defined outside its class.
It is important to understand that finalize( ) is only called just prior to garbage collection.

Using Objects as Parameters(Parameter Passing Methods):

In general, there are two ways that a computer language can pass an argument to a subroutine.

1.call-by-value: This approach copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.

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

output:
```
 a and b before call: 15 20
a and b after call: 15 20
```

2.call-by-reference: In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. The changes made to the parameter will affect the argument used to call the subroutine.

Example2:
```
class Test {
 int a, b;
Test(int i, int j) {
   a = i;
   b = j;
   }
void meth(Test o) {
o.a *=  2;
o.b /= 2;
}  }
class CallByRef
{
public static void main(String args[])
{
Test ob = new Test(15, 20);
System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b);
ob.meth(ob);
System.out.println("ob.a and ob.b after call: " + ob.a + " " + ob.b);
} }
```

Recursion:

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

A method that calls itself is said to be recursive.

Example:
```
class Factorial
 {
  int fact(int n)
    {
      if(n==1)
        return 1;
      return fact(n-1) * n;
    }
 }
```

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

Acess Control(Member Access):

Java supports rich set of access specifiers, they are:

1.public,
2.private
3.protected
4.default access level.

1.public: When a member of a class is modified by the public specifier, then that member can be accessed by any other code.

Eg: public int a;

2.private: When a member of a class is specified as private, then that member can only be accessed by other members of its class.

Eg: private int b;

3.protected: This is applied only when inheritance is used.

When the member of a class is declared as protected, then that member can be used by all the classes of same package but by only subclasses which belong to different packages.

Here package is essentially, a grouping of classes.i.e. Package acts as a container for classes and subordinate packages.

Eg: protected int c;

4.default: When no access specifier is used, then by default the member of a class is public but within its own package i.e. this member can be used by all the classes that belongs to same package.

Table: class member access

	private	No-modifier	protected	public
Same class	yes	yes	yes	yes
Same package subclass	No	yes	yes	yes
Same package non-subclass	No	yes	yes	yes
Different package subclass	No	No	yes	yes
Different package non-subclass	No	No	No	yes

Example: // this program demonstrates the difference between public and private.

```
class Test
{
int a; // default access
public int b; // public access
private int c; // private access
void setc(int i) { // set c's value
c = i;
}
int getc() { // get c's value
return c;
}  }
class AccessTest
{
public static void main(String args[])
{
Test ob = new Test();
ob.a = 10;
ob.b = 20;
//  ob.c = 100; // Error!  // This is not OK ,c is a private variable
ob.setc(100); // OK  // You must access c through its methods
System.out.println("a, b, and c: " + ob.a + " " +
ob.b + " " + ob.getc());
}  }
```

Nested and Inner classes:

It is possible to define a class within another class; such classes are known as nested classes.

The scope of a nested class is bounded by the scope of its enclosing class. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.

There are two types of nested classes: static and non-static. A static nested class is one that has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

The most important type of nested class is the inner class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

Example:// Demonstrate an inner class.

```
class Outer
 {
int outer_x = 100;
void test()
{
Inner inner = new Inner();
inner.display();
}
// this is an inner class
class Inner
{
void display()
{
System.out.println("display: outer_x = " + outer_x);
} }
}
class InnerClassDemo
{
public static void main(String args[])
 {
Outer outer = new Outer();
outer.test();
}  }
```

Exploring String class:

String is the most commonly used class in Java's class library.

The first thing to understand about strings is that every string you create is actually an object of type String. Even string constants are actually String objects.

For example, in the statement

System.out.println("This is a String, too");

Here "This is a String, too" is a String constant.

The second thing to understand about strings is that objects of type String are immutable;once a String object is created, its contents cannot be altered. While this may seem like a serious restriction, it is not, for two reasons:

• If you need to change a string, you can always create a new one that contains the modifications.
• Java defines a peer class of String, called StringBuffer, which allows strings to be altered.

Strings can be constructed in a variety of ways. The easiest is to use a statement like this:

String myString = "this is a test";

this statement displays myString: System.out.println(myString);

the operator + is used to concatenate two strings.

For example,
String myString = "I" + " like " + "Java.";

Example:// Demonstrating Strings.

```
class StringDemo
{
public static void main(String args[])
{
String strOb1 = "First String";
String strOb2 = "Second String";
String strOb3 = strOb1 + " and " + strOb2;
System.out.println(strOb1);
System.out.println(strOb2);
System.out.println(strOb3);
}
}
```

The String class contains several methods that you can use. Few of them are:

equals( ):using this method we can compare the equality of two strings.

length( ) : You can obtain the length of a string by calling the length( ) method.

charAt() : You can obtain the character at a specified index within a string by calling charAt( ).

Syntax:

```
  boolean equals(String object)
  int length( )
  char charAt(int index)
```

Example:// Demonstrating some String methods.

```
class StringDemo2
 {
public static void main(String args[])
{
String strOb1 = "First String";
String strOb2 = "Second String";
String strOb3 = strOb1;
System.out.println("Length of strOb1: " + strOb1.length());
System.out.println("Char at index 3 in strOb1: " + strOb1.charAt(3));
if(strOb1.equals(strOb2))
System.out.println("strOb1 == strOb2");
else
System.out.println("strOb1 != strOb2");
if(strOb1.equals(strOb3))
System.out.println("strOb1 == strOb3");
else
System.out.println("strOb1 != strOb3");
}
}
```

UNIT-2

Inheritance:

Inheritance in Java is a mechanism in which one class acquires all the properties and behaviors of a parent class.

- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

Syntax of Java Inheritance

```
class Subclassname extends Superclassname
   {
      //methods and fields
   }
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

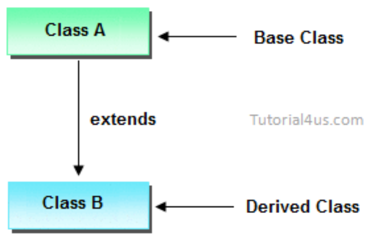
Types of inheritance in java

Based on number of ways inheriting the feature of base class into derived class we have five types of inheritance; they are:

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

1.Single inheritance

In single inheritance there exists single base class and single derived class.

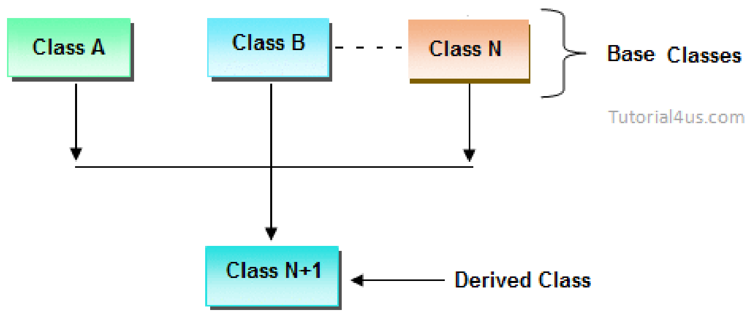


Example:

```
class Animal
{
void eat()
{
System.out.println("eating...");}
}
class Dog extends Animal
{
void bark(){
System.out.println("barking...");
}}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

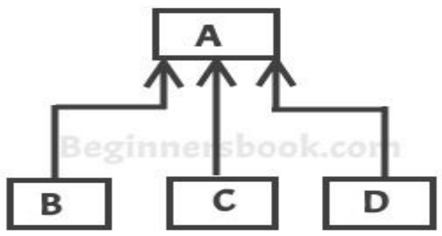
2.Multiple inheritance

In multiple inheritance there exist multiple classes and singel derived class.
Java does not implement Multiple inheritance directly but it makes use of the concept called interfaces to implement the multiple inheritance.



3.Hierarchical inheritance

In Hierarchical inheritance there exists one base class and multiple derived classes



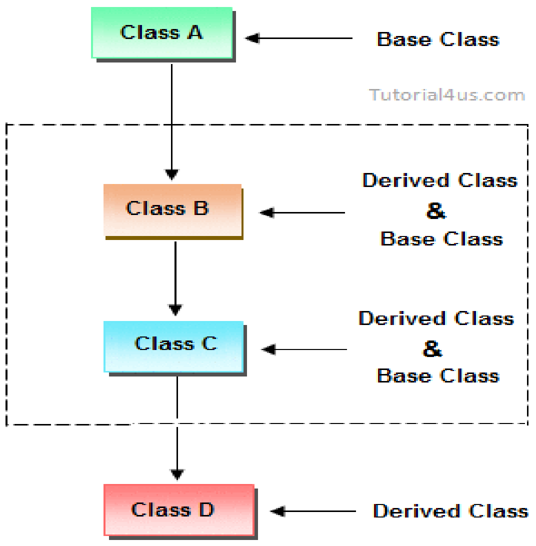
Hierarchical Inheritance

Example:

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
}}
```

4. Multilevel inheritance

In Multilevel inheritances there exists single base class, single derived class and multiple intermediate base classes.



Example:

```
class Animal{
void eat(){System.out.println("eating...");}
}
```

```
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

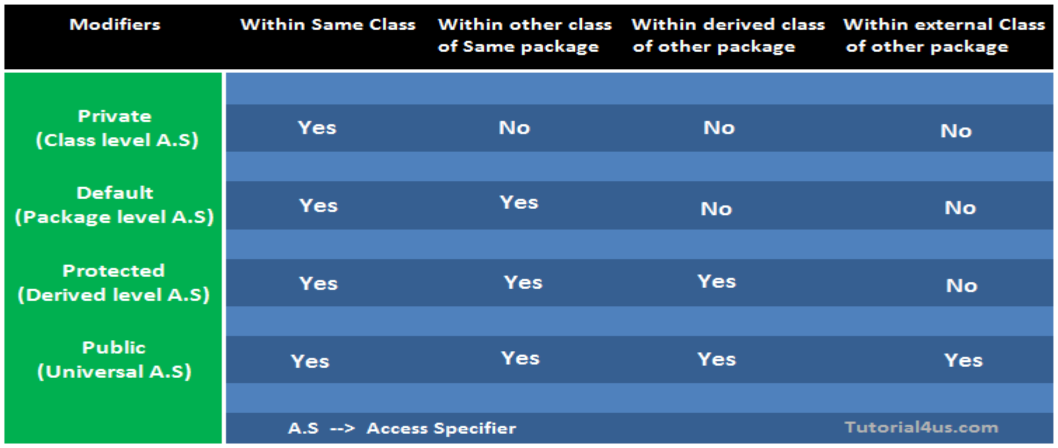
5. Hybrid inheritance

Combination of any inheritance type

Member Access Rules

Access modifiers are those which are applied before data members or methods of a class. These are used to where to access and where not to access the data members or methods. In Java programming these are classified into four types:

1. Private
2. Default (not a keyword)
3. Protected
4. Public



Modifiers	Within Same Class	Within other class of Same package	Within derived class of other package	Within external Class of other package
Private (Class level A.S)	Yes	No	No	No
Default (Package level A.S)	Yes	Yes	No	No
Protected (Derived level A.S)	Yes	Yes	Yes	No
Public (Universal A.S)	Yes	Yes	Yes	Yes

A.S --> Access Specifier

Tutorial4us.com

1.private: Private members of class in not accessible anywhere in program these are only accessible within the class. Private are also called class level access modifiers.

Example

```
class Hello
{
private int a=20;
private void show(){
System.out.println("Hello java");
}  }
 public class Demo
{
 public static void main(String args[])
 {
  Hello obj=new Hello();
  System.out.println(obj.a); //Compile Time Error, you can't access private data
  obj.show();   //Compile Time Error, you can't access private methods
 }
}
```

2.public: Public members of any class are accessible anywhere in the program in the same class and outside of class, within the same package and outside of the package. Public are also called universal access modifiers.

Example

```
class Hello
{
public int a=20;
public void show(){
System.out.println("Hello java");
}  }
public class Demo
{
 public static void main(String args[])
 {
  Hello obj=new Hello();
  System.out.println(obj.a);
  obj.show();
 }}
```

3.protected: Protected members of the class are accessible within the same class and another class of the same package and also accessible in inherited class of another package. Protected are also called derived level access modifiers.

In below the example we have created two packages pack1 and pack2. In pack1, class A is public so we can access this class outside of pack1 but method show is declared as a protected so it is only accessible outside of package pack1 only through inheritance.

Example

```
// save A.java
package pack1;
public class A
{
protected void show(){
System.out.println("Hello Java");
}  }
//save B.java
package pack2;
import pack1.*;
class B extends A
{
 public static void main(String args[]){
 B obj = new B();
 obj.show();
  }
}
```

4.default: Default members of the class are accessible only within the same class and another class of the same package. The default are also called package level access modifiers.

Example

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

```
package pack2;
import pack1.*;
class B
{
  public static void main(String args[])
 {
  A obj = new A(); //Compile Time Error, can't access outside the package
  obj.show();   //Compile Time Error, can't access outside the package
 }  }
```

Super Keyword

Super keyword in java is a reference variable that is used to refer parent class object.

1.Super keyword At Variable Level

2.Super keyword At Method Level

1.Super keyword At Variable Level

Whenever the derived class inherit base class data members there is a possibility that base class data member are similar to derived class data member and JVM gets an ambiguity.

In order to differentiate between the data member of base class and derived class, in the context of derived class the base class data members must be preceded by super keyword.

Syntax

Super.baseclass_datamember_name;

Example:

```
class Employee
{
float salary=10000;
}
class HR extends Employee
{
float salary=20000;
void display()
{
System.out.println("Salary: "+super.salary);//print base class salary
}}
class Supervarible{
public static void main(String[] args){
HR obj=new HR();
obj.display();
}}
```

2.Super keyword At Method Level

The **super keyword** can also be used to invoke or call parent class method. It should be use in case of method overriding. In other word **super keyword** use when base class method name and derived class method name have same name.

Example:

```
class Student
{
void message(){
System.out.println("Good Morning Sir");
}}
class Faculty extends Student
{
void message(){
System.out.println("Good Morning Students");
}
void display()
{
message();//will call current class message() method
super.message();//will call parent class message() method
}
public static void main(String args[])
{
Student s=new Student();
s.display();  }}
```

Using final with Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with final. Declaring a class as final implicitly declares all of its methods as final, too.

Example:

```
final class A
```

```
{
// ...
}

// The following class is illegal.
class B extends A
 {
 // ERROR! Can't subclass A
// ...  }
```

Example:

```
final class Employee
{
int salary=10000;
}
class Developer extends Employee
{
void show()
{
System.out.println("Hello Good Morning");
}
}
class FinalDemo
{
public static void main(String args[])
{
Developer obj=new Developer();
Developer obj=new Developer();
obj.show();
}
}
```

Output: Error

Object Class:

There is one special class defined in java library called, Object. All other classes are subclasses of Object. That is, Object is a superclass of all other classes.
Object defines the following methods, which means that they are available in every object.

Method	Purpose
Object clone(	Creates a new object that is the same as the object being cloned.
void finalize( )	Called before an unused object is recycled.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( )	Waits on another thread of execution.
void wait(long milliseconds)	
void wait(long milliseconds, int nanoseconds)	

Polymorphism:

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

Example:

```
class Animal{
 void eat(){System.out.println("animal is eating...");}
}

class Dog extends Animal{
 void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Animal a=new Dog();
  a.eat();
 }
}
```

In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.So compiler doesn't know its type, only its base type.

Method Overriding:

Whenever same method name is existing in both base class and derived class with same types of parameters or same order of parameters is known as **method Overriding**.

Note: Without Inheritance method overriding is not possible.

Example:

```
class Walking
{
void walk()
{
System.out.println("Man walking fastly");
}
}
class Man extends walking
{
void walk()
{
System.out.println("Man walking slowly");
}
}
class OverridingDemo
{
public static void main(String args[])
{
Man obj = new Man();
obj.walk();
}}
```

Output: Man walking slowly

Abstract Classes And Methods

In Java programming we have two types of classes they are

1.Concrete class

2.Abstract class

1.Concrete class in Java

A concrete class is one which is containing fully defined methods or implemented method.

Example

```
class  Helloworld
{
void  display()
{
System.out.println("Good Morning");
}}
```

2.Abstract class in Java

A class that is declared with abstract keyword, is known as **abstract class**. An abstract class is one which is containing some defined method and some undefined method. In java programming undefined methods are known as un-Implemented, or abstract method.

Syntax

```
abstract class className
{
......
}
```

If any class have any abstract method then that class become an abstract class.

Example

```
class Vachile
{
abstract void Bike();
 }
```

Class Vachile is become an abstract class because it have abstract Bike() method.

Abstract method

An abstract method is one which contains only declaration or prototype but it never contains body or definition. In order to make any undefined method as abstract whose declaration is must be predefined by abstract keyword.

Syntax:

abstract ReturnType methodName(List of formal parameter);

Example:

```
abstract   void  sum();
abstract  void  diff(int, int);
```

Example:

```
abstract class Vachile
{
 abstract void speed();
}
class Bike extends Vachile
{
void speed()
{
System.out.println("Speed limit is 40 km/hr..");
```

```
 }
 public static void main(String args[])
 {
  Vachile obj = new Bike(); //indirect object creation
  obj.speed();
  }  }
```

Note: An object of abstract class cannot be created directly, but it can be created indirectly. It means you can create an object of abstract derived class. You can see in above example

Example

```
Vachile obj = new Bike(); //indirect object creation
```

Interfaces

Interface is similar to class which is collection of public static final variables (constants) and abstract methods.

The interface is a mechanism to achieve fully abstraction in java. There can be only abstract methods in the interface. It is used to achieve fully abstraction and multiple inheritance in Java.

Declaring Interfaces

The **interface** keyword is used to declare an interface.

Example

```
interface Person
{
 datatype variablename=value;
 returntype methodname(parameters list);
}
```

Implementing Interfaces:

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

Example:

```
interface Person
{
void run();  // abstract method
}
class A implements Person
{
public void run()
{
System.out.println("Run fast");
}
public static void main(String args[])
 {
 A obj = new A();
 obj.run();
 }
}
```

Difference between Abstract class and Interface

Abstract class	Interface
It is collection of abstract method and concrete methods.	It is collection of abstract method.
There properties can be reused commonly in a specific application.	There properties commonly usable in any application of java environment.
It does not support multiple inheritance.	It support multiple inheritance.
Abstract class is preceded by abstract keyword.	It is preceded by Interface keyword.
Which may contain either variable or constants.	Which should contains only constants.
The default access specifier of abstract class methods are default.	There default access specifier of interface method are public.
These class properties can be reused in other class using extend keyword.	These properties can be reused in any other class using implements keyword.
Inside abstract class we can take constructor.	Inside interface we can not take any constructor.
For the abstract class there is no restriction like initialization of variable at the time of variable declaration.	For the interface it should be compulsory to initialization of variable at the time of variable declaration.
There are no any restriction for abstract class variable.	For the interface variable can not declare variable as private, protected, transient, volatile.
There are no any restriction for abstract class method modifier that means we can use any modifiers.	For the interface method can not declare method as strictfp, protected, static, native, private, final, synchronized.

Inner Classes

If one class is existing within another class is known as inner class or nested class

Syntax

```
class  Outerclass_name
{
.....
.....
class  Innerclass_name1
{
.....
.....
}
class  Innerclass_name1
{
.....
.....
}
.....
}
```

Uses Of Inner Classes

1.To provide more security by making those inner class properties specific to only outer class but not for external classes.

2.To make more than one property of classes private properties.

There are two types of nested classes non-static and static nested classes.The non-static nested classes are also known as inner classes.

1.Non-static nested class (inner class)
  - i)Member inner class
  - ii)Anonymous inner class
  - iii)Local inner class

2.Static nested class

Type	**Description**
Member Inner Class	A class created within class and outside method.
Anonymous Inner	A class created for implementing interface or extending class. Its

Class	name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

1.Java Member inner class

A non-static class that is created inside a class but outside a method is called member inner class.

Syntax:

```
class Outer
{
   class Inner
     {
     }
}
```

Example

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

Output:data is 30

2.Java Anonymous inner class

A class that have no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface.

Example

```
abstract class Person
```

```
    {
        abstract void eat();
    }
class TestAnonymousInner
{
 public static void main(String args[])
{
    Person p=new Person()
      {
        void eat()
         {
            System.out.println("nice fruits");
          }
        };
  p.eat();
 }
 }
```

3.Java Local inner class

A class i.e. created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Example

```
public class localInner
{
 private int data=30;
 void display()
 {
      class Local
        {
            void msg()
             {
                 System.out.println(data);
              }
          }
   Local l=new Local();
   l.msg();
 }
 public static void main(String args[]){
  localInner obj=new localInner();
  obj.display();
 }
}
```

Static Nested Class

A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.

- It can access static data members of outer class including private.
- Static nested class cannot access non-static (instance) data member or method.

Example

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

Package in Java

A package is a collection of similar types of classes, interfaces and sub-packages.

Type of package

Package are classified into two type which are given below.

1. Predefined or built-in package
2. User defined package

1.Predefined or built-in package

These are the package which are already designed by the Sun Microsystem and supply as a part of java API, every predefined package is collection of predefined classes, interfaces and sub-package.

- **java.lang** – bundles the fundamental classes
- **java.io** – classes for input , output functions are bundled in this package

2.User defined package

If any package is design by the user is known as user defined package. User defined package are those which are developed by java programmer and supply as a part of their project to deal with common requirement.

Creating a package:

To create a package include a package command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package.
If you omit the package statement, the class names are put into the default package, which has no name.

Syntax: package pkg;
Here, pkg is the name of the package.
Example: *package mypackage;*

We can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period.
Syntax: package pkg1[.pkg2[.pkg3]];
Eg: package java.awt.image;

Example:

```
package mypackage;
public class A
{
public void show()
{
System.out.println("Sum method");
}
}
```

Importing packages:

Java includes the *import* statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name.

```
import mypackage.A;
public class Hello
{
public static void main(String arg[])
{
A a=new A();
a.show();
System.out.println("show() class A");
}
}
```

Unit-3

Exception Handling in Java

The process of converting system error messages into user friendly error message is known as **Exception handling**. This is one of the powerful feature of Java to handle run time error and maintain normal flow of java application.

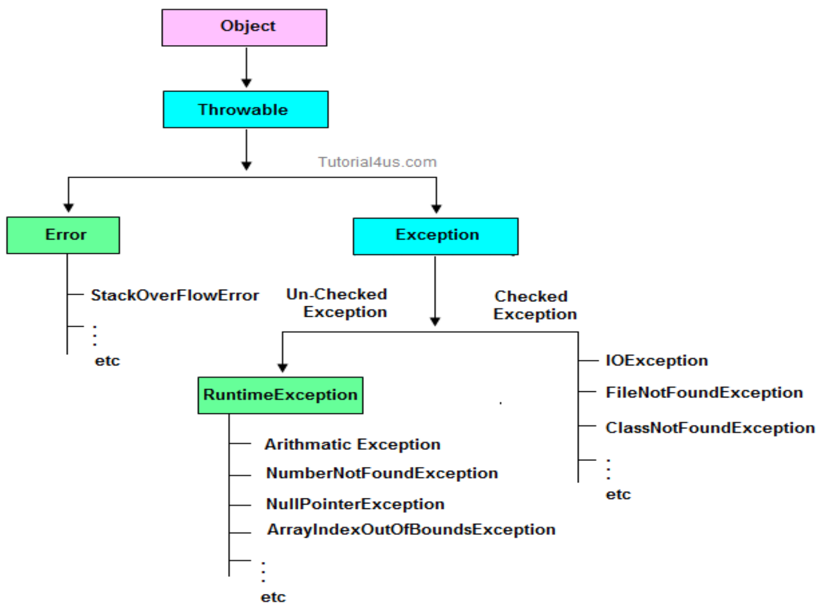
Exception

An **Exception** is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's Instructions.

Why use Exception Handling

Handling the exception is nothing but converting system error generated message into user friendly error message. Whenever an exception occurs in the java application, JVM will create an object of appropriate exception of sub class and generates system error message, these system generated messages are not understandable by user so need to convert it into user friendly error message. You can convert system error message into user friendly error message by using exception handling feature of java.
For Example: when you divide any number by zero then system generate **/ by zero** so this is not understandable by user so you can convert this message into user friendly error message like **Don't enter zero for denominator.**

Hierarchy of Exception classes



Type of Exception

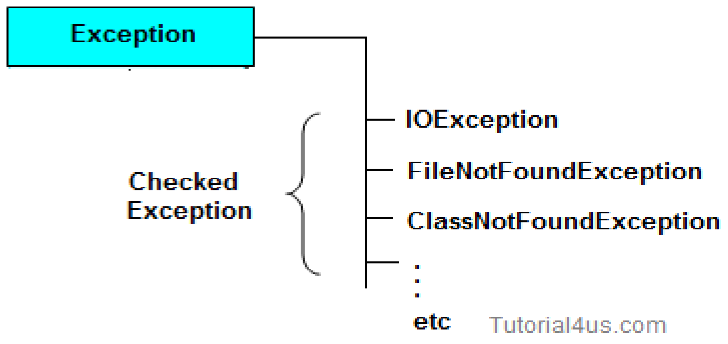
1.Checked Exception

2.Un-Checked Exception

1.Checked Exception

Checked Exception are the exception which checked at compile-time. These exception are directly sub-class of java.lang.Exception class.

Only for remember: Checked means checked by compiler so checked exception are checked at compile-time.

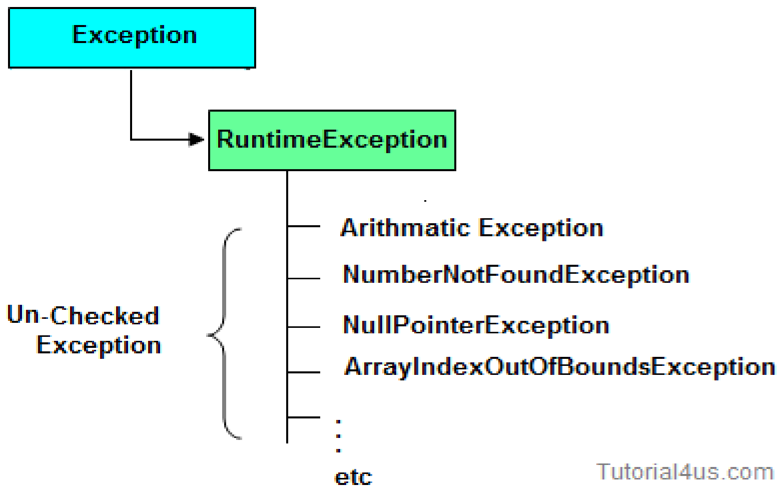


2.Un-Checked Exception

Un-Checked Exception are the exception both identifies or raised at run time. These exception are directly sub-class of java.lang.RuntimeException class.

Note: In real time application mostly we can handle un-checked exception.

Only for remember: Un-checked means not checked by compiler so un-checked exception are checked at run-time not compile time.



Difference between checked Exception and un-checked Exception

	Checked Exception	Un-Checked Exception
1	checked Exception are checked at compile time	un-checked Exception are checked at run time
3	e.g. FileNotFoundException, NumberNotFoundException etc.	e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

Difference between Error and Exception

	Error	Exception
1	Can't be handle.	Can be handle.
2	Example: NoSuchMethodError OutOfMemoryError	Example: ClassNotFoundException NumberFormateException

Uncaught Exceptions(with out using try&catch):

Example without Exception Handling

```
class ExceptionDemo
{
public static void main(String[] args)
{
int a=10, ans=0;
ans=a/0;
System.out.println("Denominator not be zero");
}
```

}

Abnormally terminate program and give a message like below, this error message is not understandable by user so we convert this error message into user friendly error message, like "denominator not be zero".

Handling the Exception

Handling the exception is nothing but converting system error generated message into user friendly error message in others word whenever an exception occurs in the java application, JVM will create an object of appropriate exception of sub class and generates system error message, these system generated messages are not understandable by user so need to convert it into user-friendly error message. You can convert system error message into user-friendly error message by using exception handling feature of java.

Use Five keywords for Handling the Exception

1.try

2.catch

3.finally

4.throws

5.throw

Syntax for handling the exception:

```
try
{
  // statements causes problem at run time
}
catch(type of exception-1 object-1)
{
  // statements provides user friendly error message
}
catch(type of exception-2 object-2)
{
  // statements provides user friendly error message
}
finally
{
  // statements which will execute compulsory
}
```

1.try block

Inside **try block** we write the block of statements which causes executions at run time in other words try block always contains problematic statements.

Important points about try block

- If any exception occurs in try block then CPU controls comes out to the try block and executes appropriate catch block.
- After executing appropriate catch block, even through we use run time statement, CPU control never goes to try block to execute the rest of the statements.
- Each and every try block must be immediately followed by catch block that is no intermediate statements are allowed between try and catch block.
- Each and every try block must contains at least one catch block. But it is highly recommended to write multiple catch blocks for generating multiple user friendly error messages.
- One try block can contains another try block that is nested or inner try block can be possible.

2.catch block

Inside **catch** block we write the block of statements which will generates user friendly error messages.

catch block important points

- Catch block will execute exception occurs in try block.
- You can write multiple catch blocks for generating multiple user friendly error messages to make your application strong. You can see below example.
- At a time only one catch block will execute out of multiple catch blocks.
- in catch block you declare an object of sub class and it will be internally referenced by JVM.

Example:(try&catch):

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

```
int a=10, ans=0;
try
{
ans=a/0;
}
catch (Exception e)
{
System.out.println("Denominator not be zero");
}
}}
```

Output

```
Denominator not be zero
```

3.throw

throw is a keyword in java language which is used to throw any user defined exception to the same signature of method in which the exception is raised.

Note: throw keyword always should exist within method body.

whenever method body contain throw keyword than the call method should be followed by throws keyword.

Syntax

```
class className
{
returntype method(...) throws Exception_class
{
throw(Exception obj)
}
}
```

4.throws

throws is a keyword in java language which is used to throw the exception which is raised in the called method to it's calling method throws keyword always followed by method signature.

Example

```
returnType methodName(parameter)throws Exception_class....
{
.....
}
```

Difference between throw and throws

	throw	throws
1	throw is a keyword used for hitting and generating the exception which are occurring as a part of method body	throws is a keyword which gives an indication to the specific method to place the common exception methods as a part of try and catch block for generating user friendly error messages
2	The place of using throw keyword is always as a part of method body.	The place of using throws is a keyword is always as a part of method heading
3	When we use throw keyword as a part of method body, it is mandatory to the java programmer to write throws keyword as a part of method heading	When we write throws keyword as a part of method heading, it is optional to the java programmer to write throw keyword as a part of method body.

Example of throw and throws:

Example

```
package pack;
public class DivZero
{
public void division(int a, int b)throws ArithmeticException
{
if(b==0)
{
ArithmeticException ae=new ArithmeticException("Does not enter zero for Denominator");
throw ae;
}
else
{
int c=a/b;
System.out.println("Result: "+c);
}}}
```

5.finally Block

Inside finallyblock we write the block of statements which will relinquish (released or close or terminate) the resource (file or database) where data store permanently.

finally block important points

- Finally block will execute compulsory
- Writing finally block is optional.
- You can write finally block for the entire java program
- In some of the circumstances one can also write try and catch block in finally block.

Example

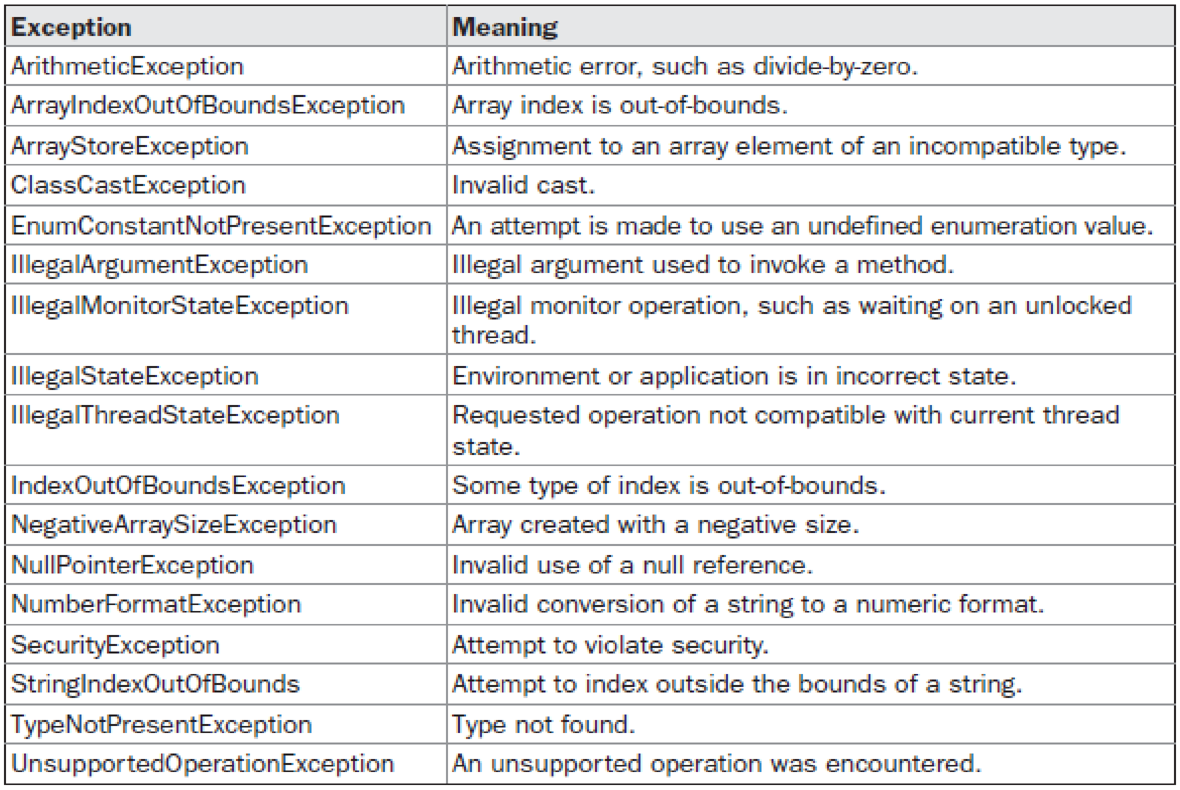
```
class ExceptionDemo
{
public static void main(String[] args)
{
int a=10, ans=0;
try
{
ans=a/0;
}
catch (Exception e)
{
System.out.println("Denominator not be zero");
}
finally
{
System.out.println("I am from finally block");
}}}
```

Output

```
Denominator not be zero
I am from finally block
```

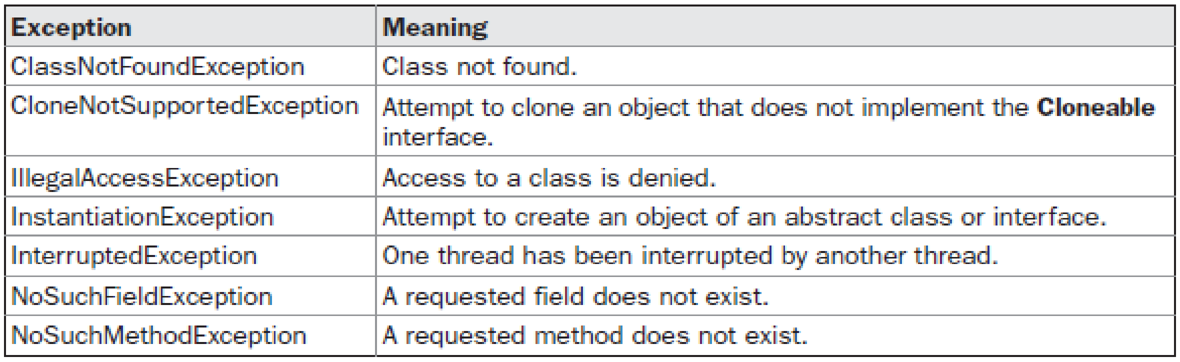
Java’s Built-in Exceptions

Inside the standard package **java.lang**, Java defines several exception classes.



Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

TABLE 10-1 Java’s Unchecked **RuntimeException** Subclasses Defined in **java.lang**



Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

TABLE 10-2 Java’s Checked Exceptions Defined in **java.lang**

Custom Exception in Java

If any exception is design by the user known as user defined or Custom Exception. Custom Exception is created by user.

Rules to design user defined Exception

1. Create a package with valid user defined name.
2. Create any user defined class.
3. Make that user defined class as derived class of Exception or RuntimeException class.
4. Declare parametrized constructor with string variable.
5. call super class constructor by passing string variable within the derived class constructor.
6. Save the program with public class name.java

Example

```
package nage;
public class AgeException extends Exception
{
public AgeException(String s)
{
super(s);
}
}
```

Benefits of Exception Handling

- In java, Exceptional Handling is a very good technique to handle run time errors in the program.
- Exception handling helps us catch or identify abnormal scenarios in our code.
- Exception causes abnormal termination of currently executing program. We can avoid the abnormal termination of the program by handling the exception using the keywords throw, throws, try, catch and finally.
- It will help us to display messages for the end-users about the behavior of the program.
- It separates the Error Handling Code from "Regular" Code.
- Good exception handling framework helps an application to run smoothly.

- Basically exception handling helps an application to maintain its normal flow. Even if some unexpected error occurs, the exception framework provides separate execution path to avoid application failure

Multithreading in Java

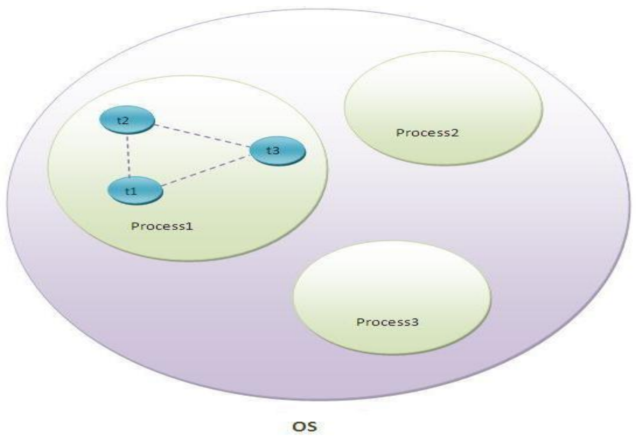
Multithreading in java is a process of executing multiple threads simultaneously. The aim of multithreading is to achieve the concurrent execution.

Differences Between Multiprocessing and Multithreading:

Multiprocessing	**Multithreading**
1.Each process have its own address in memory i.e. each process allocates separate memory area.	1.Threads share the same address space.
2.Process is heavyweight.	2.Thread is lightweight.
3.Cost of communication between the process is high.	3.Cost of communication between the thread is low.
4.Process-based multitasking is totally controlled by the operating system.	4.thread-based multitasking can be controlled by the programmer to some extent in a program.

What is Thread?

A thread is a lightweight subprocess, a smallest unit of processing. It is a separate path of execution. It shares the memory area of process.



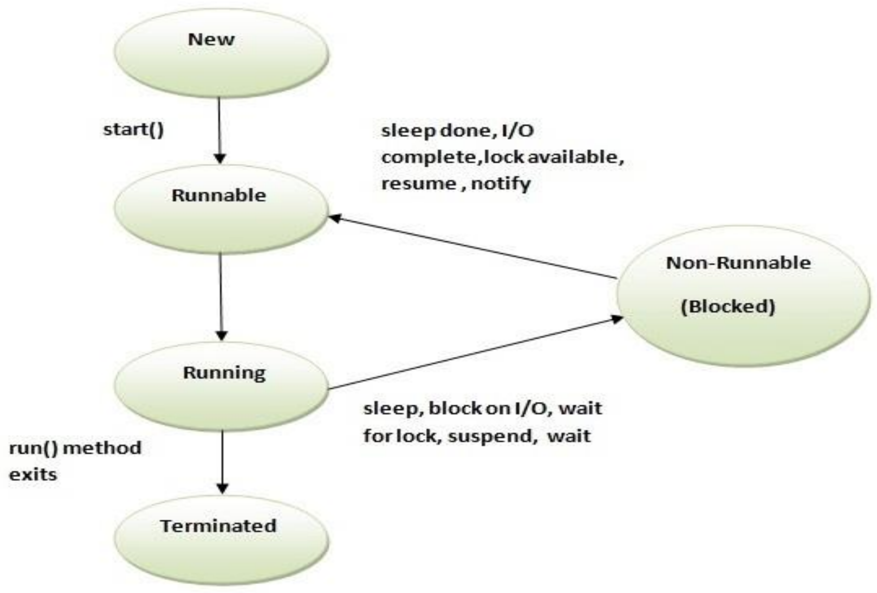
As shown in the above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

Note: At a time only one thread is executed.

Life cycle of a Thread (Thread States):

A thread can be in one of the five states in the thread. The life cycle of the thread is controlled by JVM. The thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated


New
start()
sleep done, I/O complete, lock available, resume , notify
Runnable
Non-Runnable (Blocked)
Running
sleep, block on I/O, wait for lock, suspend, wait
run() method exits
Terminated

1)New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2)Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3)Running

The thread is in running state if the thread scheduler has selected it.

4)Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5)Terminated

A thread is in terminated or dead state when its run() method exits.

How to create thread:

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

1.Thread class:

Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).

16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.

2.Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

public void run(): is used to perform action for a thread

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.

When the thread gets a chance to execute, its target run() method will run.

1)By extending Thread class:

```
class Multi extends Thread
{
public void run()
{
System.out.println("thread is running...");
}
public static void main(String args[])
{
Multi t1=new Multi();
t1.start();
 }
}
```

Output:thread is running...

2)By implementing the Runnable interface:

```
class Multi3 implements Runnable
{
public void run()
{
```

```
System.out.println("thread is running...");
}
public static void main(String args[])
{
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

Output:thread is running...

If you are not extending the Thread class,your class object would not be treated as a thread object.So you need to explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute.

Priority of a Thread (Thread Priority):

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defiend in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

```
class Multi10 extends Thread{
 public void run()
{
   System.out.println("running thread name is:"+Thread.currentThread().getName());
   System.out.println("running thread priority is:"+Thread.currentThread().getPriority());

  }
 public static void main(String args[])
{
  Multi10 m1=new Multi10();
  Multi10 m2=new Multi10();
```

```
    m1.setPriority(Thread.MIN_PRIORITY);
    m2.setPriority(Thread.MAX_PRIORITY);
    m1.start();
    m2.start();

   }
  }
```

Output:

```
running thread name is:Thread-0
running thread priority is:10
running thread name is:Thread-1
running thread priority is:1
```

Synchronization:

Synchronization is the capability of control the access of multiple threads to any shared resource. Synchronization is better in case we want only one thread can access the shared resource at a time.

Why use Synchronization?

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. Static synchronization.
2. Cooperation (Inter-thread communication in java)

Java Synchronized Method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Example:

```
Class Table
{
  synchronized void printTable(int n){
  for(int i=1;i<=5;i++){
    System.out.println(n*i);
    try{
     Thread.sleep(400);
    }catch(Exception e){System.out.println(e);}
  }   }  }
class MyThread1 extends Thread
{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}
}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}  }
class Use{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Output: 5

```
10
15
20
25
100
200
300
400
500
```

Synchronized Block in Java

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose we have 50 lines of code in our method, but we want to synchronize only 5 lines, in such cases, we can use synchronized block.

If we put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Syntax

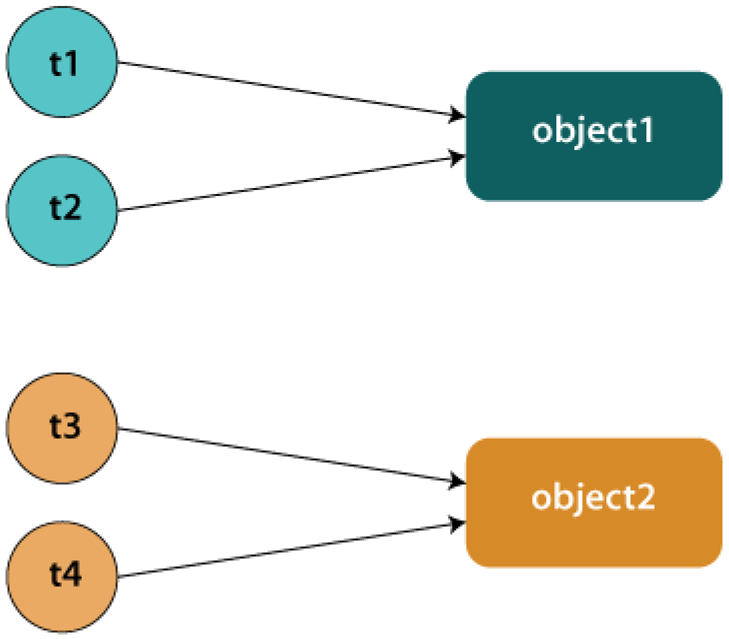
```
synchronized (object reference expression) {
  //code block
}
```

Example:

```
class Table
{
 void printTable(int n){
   synchronized(this){//synchronized block
     for(int i=1;i<=5;i++){
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
 }//end of the method
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

public class TestSynchronizedBlock1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Static Synchronization

If you make any static method as synchronized, the lock will be on the class not on object.



Problem without static synchronization

Suppose there are two objects of a shared class (e.g. Table) named object1 and object2. In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock. But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock. We don't want interference between t1 and t3 or t2 and t4. Static synchronization solves this problem.

Example of Static Synchronization

In this example we have used **synchronized** keyword on the static method to perform static synchronization.

TestSynchronization4.java

```
class Table
{
 synchronized static void printTable(int n){
   for(int i=1;i<=10;i++){
     System.out.println(n*i);
     try{
       Thread.sleep(400);
     }catch(Exception e){}
   }
 }
}
class MyThread1 extends Thread{
public void run(){
Table.printTable(1);
}
}
class MyThread2 extends Thread{
public void run(){
Table.printTable(10);
}
}
class MyThread3 extends Thread{
public void run(){
Table.printTable(100);
}
}
class MyThread4 extends Thread{
public void run(){
Table.printTable(1000);
}
}
public class TestSynchronization4{
public static void main(String t[]){
MyThread1 t1=new MyThread1();
MyThread2 t2=new MyThread2();
MyThread3 t3=new MyThread3();
MyThread4 t4=new MyThread4();
t1.start();
t2.start();
t3.start();
t4.start();
}
}
```

Interthread Communication:

For example, consider the classic queuing problem ie. Producer consumer problem

case 1: Fast producer slow consumer.

Assumption: Queue can hold only one data item at one time.

To avoid this problem, Java includes an elegant inter process communication mechanism via the wait( ), notify( ), and notifyAll( ) methods. These methods are implemented as final methods in Object, so all classes have them.

wait( ) tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ).

notify( ) wakes up a thread that called wait( ) on the same object.

notifyAll( ) wakes up all the threads that called wait( ) on the same object. One of the threads will be granted access.

Syntax:

final void wait( ) throws InterruptedException

final void notify( )

final void notifyAll( )

Producer and Consumer Problem

The producer-consumer problem (also known as the bounded-buffer problem) is another classical example of a multithread synchronization problem. The problem describes two threads, the producer and the consumer, who share a common, fixed-size buffer. The producer's job is to generate a piece of data and put it into the buffer. The consumer is consuming the data from the same buffer simultaneously. The problem is to make sure that the producer will not try to add data into the buffer if it is full and that the consumer will not try to remove data from an empty buffer.The solution for this problem involves two parts. The producer should wait when it tries to put the newly created product into the buffer until there is at least one free slot in the buffer. The consumer, on the other hand, should stop consuming if the buffer is empty.

Example of a producer and consumer.

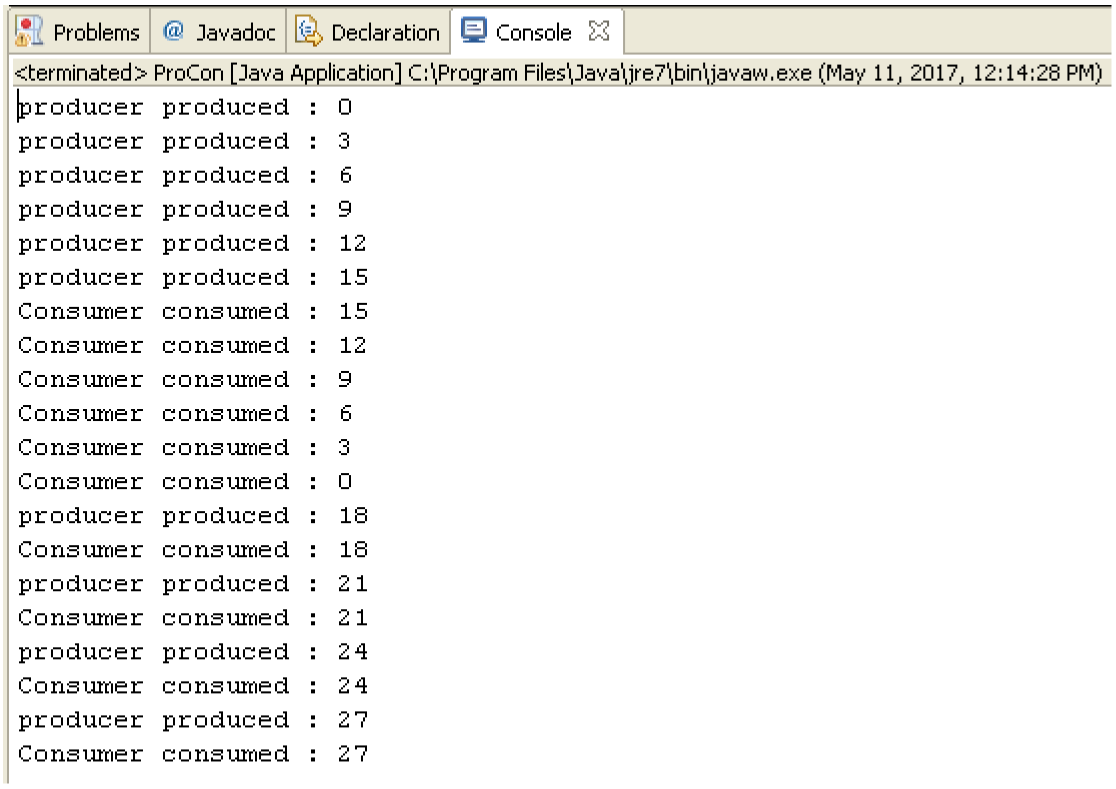
```
class Buffer
{
        int buffer[]=new int[10],ptr=0;
        synchronized void put(int value)
        {
                if(ptr==10)
                        try
                        {
                        wait();
                        }
                        catch(InterruptedException e)
                        {
                                System.out.println(e);
                        }
                buffer[ptr]=value;
                ptr++;
                notifyAll();
        }
        synchronized int get()
        {
                if(ptr==0)
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e)
                        {
                                System.out.println(e);
                        }
                        ptr--;
                        notifyAll();
                        return buffer[ptr];
        }
}
class Consumer extends Thread
{
        Buffer b;
        Consumer(Buffer b)
        {
                this.b=b;
```

```
	}
	public void run()
	{
		int value;
		for(int i=0;i<10;i++)
		{
		value=b.get();
		System.out.println("Consumer consumed : "+value);
		}
	}
}

class Producer extends Thread
{
	Buffer b;
	Producer(Buffer b)
	{
	this.b=b;
	}
	public void run()
	{
		for(int i=0;i<10;i++)
		{
		int value=i*3;
		b.put(value);
		System.out.println("producer produced : "+value);
		try
		{
			sleep(10);
		}
		catch(InterruptedException e)
		{
			System.out.println(e);
		}
		}
	}
}

class ProCon
{
	public static void main(String ar[])
	{
		Buffer b=new Buffer();
		Producer p=new Producer(b);
		Consumer c=new Consumer(b);
		p.setPriority(Thread.MAX_PRIORITY);;
```

```
                    c.setPriority(Thread.MIN_PRIORITY);
                    p.start();
                    c.start();
          }
}
```

OUTPUT:

javac ProCon.java
java ProCon



```
<terminated> ProCon [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (May 11, 2017, 12:14:28 PM)
producer produced : 0
producer produced : 3
producer produced : 6
producer produced : 9
producer produced : 12
producer produced : 15
Consumer consumed : 15
Consumer consumed : 12
Consumer consumed : 9
Consumer consumed : 6
Consumer consumed : 3
Consumer consumed : 0
producer produced : 18
Consumer consumed : 18
producer produced : 21
Consumer consumed : 21
producer produced : 24
Consumer consumed : 24
producer produced : 27
Consumer consumed : 27
```

Unit-IV
Collections

Collection:

A Collection is simply an object that groups multiple elements into a single unit.

Framework:

A Framework provides ready-made architecture and represents set of classes and interface.

Collection Framework:

It is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

- Interfaces
- Interface Implementer Classes

Where use Collection Framework

All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

Collection Framework API

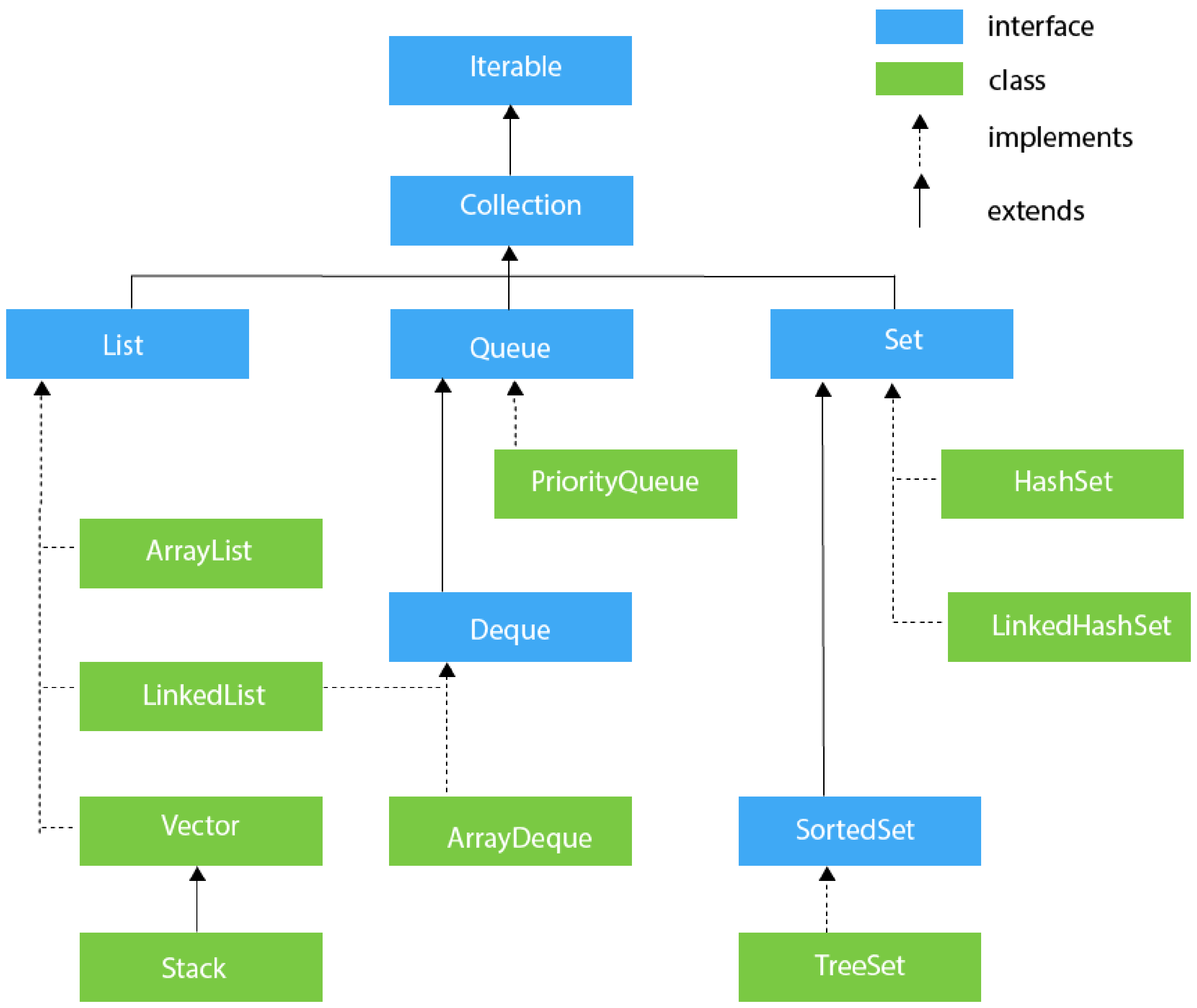
Collection Framework API is a java API that is contains all Interfaces and classes of Collection Framework which is used for perform searching, sorting, insertion, manipulation, deletion operations.

Package

java.util package

Collection Framework Hierarchy

Almost all collections in Java are derived from the java.util.Collection interface



Methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

No.	Method	Description
1	public boolean add(E e)	It is used to insert an element in this collection.
2	public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.

3	public boolean remove(Object element)	It is used to delete an element from the collection.
4	public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
5	default boolean removeIf(Predicate<? super E> filter)	It is used to delete all the elements of the collection that satisfy the specified predicate.
6	public boolean retainAll(Collection<?> c)	It is used to delete all the elements of invoking collection except the specified collection.
7	public int size()	It returns the total number of elements in the collection.
8	public void clear()	It removes the total number of elements from the collection.
9	public boolean contains(Object element)	It is used to search an element.
10	public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
11	public Iterator iterator()	It returns an iterator.
12	public Object[] toArray()	It converts collection into array.
13	public <T> T[] toArray(T[] a)	It converts collection into array. Here, the runtime type of the returned array is that of the specified array.
14	public boolean isEmpty()	It checks if collection is empty.

15	default Stream<E> parallelStream()	It returns a possibly parallel Stream with the collection as its source.
16	default Stream<E> stream()	It returns a sequential Stream with the collection as its source.
17	default Spliterator<E> spliterator()	It generates a Spliterator over the specified elements in the collection.
18	public boolean equals(Object element)	It matches two collections.
19	public int hashCode()	It returns the hash code number of the collection.

Collection Interface

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are Boolean add (Object obj), Boolean addAll (Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

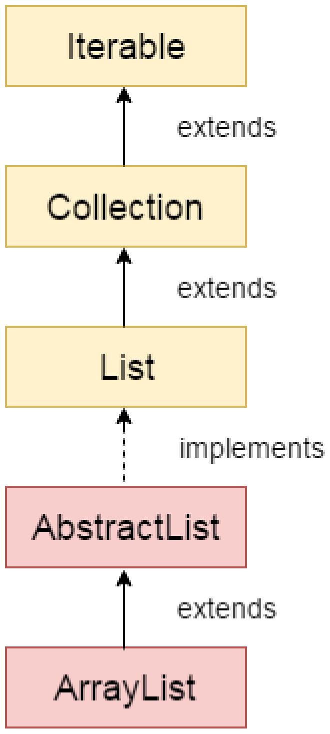
To instantiate the List interface, we must use :

1. List <data-type> list1= **new** ArrayList();
2. List <data-type> list2 = **new** LinkedList();
3. List <data-type> list3 = **new** Vector();
4. List <data-type> list4 = **new** Stack();

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

The classes that implement the List interface are given below.

ArrayList



Java **ArrayList** class uses a *dynamic array* for storing the elements. It is like an array, but there is *no size limit*. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in the *java.util* package.

The ArrayList in Java can have the duplicate elements also.

The ArrayList class implements the List interface The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

Java Non-generic Vs. Generic Collection

Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.

Java new generic collection allows you to have only one type of object in a collection. Now it is type safe so typecasting is not required at runtime.

Let's see the old non-generic example of creating java collection.

1. ArrayList list=**new** ArrayList();//creating old non-generic arraylist

Let's see the new generic example of creating java collection.

1. ArrayList<String> list=**new** ArrayList<String>();//creating new generic arraylist

In a generic collection, we specify the type in angular braces. Now ArrayList is forced to have the only specified type of objects in it. If you try to add another type of object, it gives *compile time error*.

```
import java.util.*;
class TestJavaCollection1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();//Creating arraylist
list.add("Ravi");//Adding object in arraylist
list.add("Vijay");
list.add("Ravi");
list.add("Ajay");
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ravi

Vijay

Ravi

Ajay
```

Get and Set ArrayList

The *get() method* returns the element at the specified index, whereas the *set() method* changes the element.

```
import java.util.*;
public class ArrayListExample4{
 public static void main(String args[]){
  ArrayList<String> al=new ArrayList<String>();
  al.add("Mango");
  al.add("Apple");
  al.add("Banana");
  al.add("Grapes");
  //accessing the element
  System.out.println("Returning element: "+al.get(1));//it will return the 2nd element,
because index starts from 0
  //changing the element
  al.set(1,"Dates");
  //Traversing list
  for(String fruit:al)
    System.out.println(fruit);

 }
}
```

Output:

```
Returning element: Apple
```

```
Mango

Dates

Banana

Grapes
```

Sort ArrayList

The *java.util* package provides a utility class **Collections** which has the static method sort(). Using the **Collections.sort()** method, we can easily sort the ArrayList.

User-defined class objects in Java ArrayList

Let's see an example where we are storing Student class object in an array list.

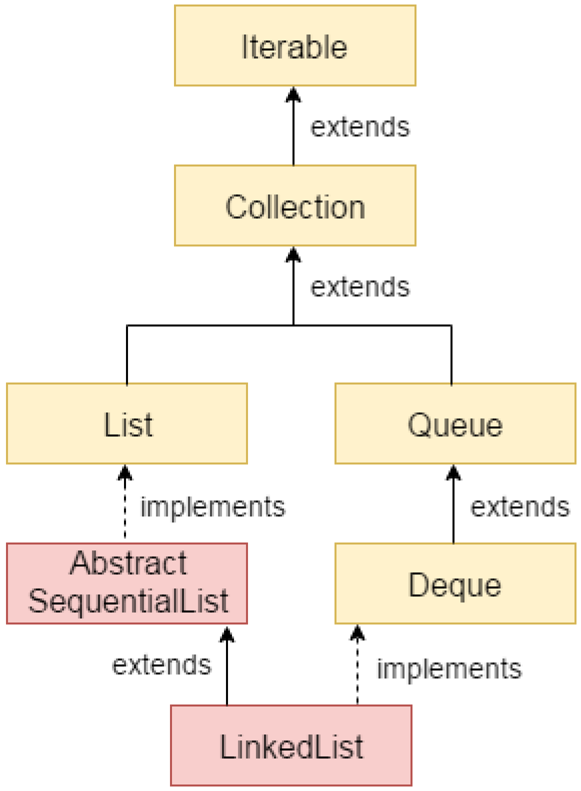
```
class Student{
  int rollno;
  String name;
  int age;
  Student(int rollno,String name,int age){
   this.rollno=rollno;
   this.name=name;
   this.age=age;
  }
}

import java.util.*;
 class ArrayList5{
 public static void main(String args[]){
  //Creating user-defined class objects
  Student s1=new Student(101,"Sonoo",23);
  Student s2=new Student(102,"Ravi",21);
  Student s2=new Student(103,"Hanumat",25);
  //creating arraylist
  ArrayList<Student> al=new ArrayList<Student>();
  al.add(s1);//adding Student class object
```

```
   al.add(s2);
   al.add(s3);
   //Getting Iterator
   Iterator itr=al.iterator();
   //traversing elements of ArrayList object
   while(itr.hasNext()){
     Student st=(Student)itr.next();
     System.out.println(st.rollno+" "+st.name+" "+st.age);
   }
  }
 }
```

Output:

```
101 Sonoo 23

102 Ravi 21

103 Hanumat 25
```

Java LinkedList class



Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- Java LinkedList class is non synchronized.
- In Java LinkedList class, manipulation is fast because no shifting needs to occur.
- Java LinkedList class can be used as a list, stack or queue.

Hierarchy of LinkedList class

As shown in the above diagram, Java LinkedList class extends AbstractSequentialList class and implements List and Deque interfaces.

Doubly Linked List

In the case of a doubly linked list, we can add or remove elements from both sides.

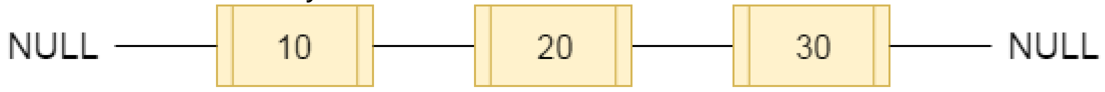


fig- doubly linked list

LinkedList class declaration

Let's see the declaration for java.util.LinkedList class.

1. **public class** LinkedList<E> **extends** AbstractSequentialList<E> **implements** List<E>, Deque<E>, Cloneable, Serializable

Constructors of Java LinkedList

Constructor	Description
LinkedList()	It is used to construct an empty list.
LinkedList(Collection<E> c)	It is used to construct a list containing the elements of the specified collection

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Methods:

Method	Description
boolean add(E e)	It is used to append the specified element to the end of a list.
void add(int index, E element)	It is used to insert the specified element at the specified position index in a list.
boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean addAll(int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void addFirst(E e)	It is used to insert the given element at the beginning of a list.
void addLast(E e)	It is used to append the given element to the end of a list.
void clear()	It is used to remove all the elements from a list.
Object clone()	It is used to return a shallow copy of an ArrayList.
boolean contains(Object o)	It is used to return true if a list contains a specified element.
Iterator<E> descendingIterator()	It is used to return an iterator over the elements in a deque in reverse sequential order.
E element()	It is used to retrieve the first element of a list.
E get(int index)	It is used to return the element at the specified position in a list.

E getFirst()	It is used to return the first element in a list.
E getLast()	It is used to return the last element in a list.
int indexOf(Object o)	It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element.
int lastIndexOf(Object o)	It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element.
ListIterator<E> listIterator(int index)	It is used to return a list-iterator of the elements in proper sequence, starting at the specified position in the list.
boolean offer(E e)	It adds the specified element as the last element of a list.
boolean offerFirst(E e)	It inserts the specified element at the front of a list.
boolean offerLast(E e)	It inserts the specified element at the end of a list.
E peek()	It retrieves the first element of a list
E peekFirst()	It retrieves the first element of a list or returns null if a list is empty.
E peekLast()	It retrieves the last element of a list or returns null if a list is empty.
E poll()	It retrieves and removes the first element of a list.
E pollFirst()	It retrieves and removes the first element of a list, or returns null if a list is empty.
E pollLast()	It retrieves and removes the last element of a list, or returns null if a list is empty.
E pop()	It pops an element from the stack represented by a list.

void push(E e)	It pushes an element onto the stack represented by a list.
E remove()	It is used to retrieve and removes the first element of a list.
E remove(int index)	It is used to remove the element at the specified position in a list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element in a list.
E removeFirst()	It removes and returns the first element from a list.
boolean removeFirstOccurrence(Object o)	It is used to remove the first occurrence of the specified element in a list (when traversing the list from head to tail).
E removeLast()	It removes and returns the last element from a list.
boolean removeLastOccurrence(Object o)	It removes the last occurrence of the specified element in a list (when traversing the list from head to tail).
E set(int index, E element)	It replaces the element at the specified position in a list with the specified element.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection2{
public static void main(String args[]){
LinkedList<String> al=new LinkedList<String>();
al.add("Ravi");
al.add("Vijay");
al.add("Ravi");
al.add("Ajay");
Iterator<String> itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ravi

Vijay

Ravi

Ajay
```

Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection3{
public static void main(String args[]){
Vector<String> v=new Vector<String>();
v.add("Ayush");
v.add("Amit");
v.add("Ashish");
v.add("Garima");
Iterator<String> itr=v.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Data Retrieving Technique form Collection Framework
Technique to retrieve elements from Collection object

Java support following technique to retrieve the elements from any collection object.

1. Iterator interface
2. Enumeration interface

1.Iterator Interface
Methods:
public boolean hasNext()

This method return true provided by Iterator interface object is having next element otherwise it returns false.

public object next()

This method is used for retrieving next element of any collection framework variable provided public boolean hasNext(). If next elements are available then this method returns true other wise it return false.

Example:

```
import java.util.*;
class IteratorDemo
{
public static void main(String args[])
{
ArrayList<Integer> al=new ArrayList<Integer>();  // creating arraylist
al.add(10);
al.add(20);
al.add(30);
Iterator itr=al.iterator();  // getting Iterator from arraylist to traverse elements
while(itr.hasNext())
{
System.out.println(itr.next());
  }
 }  }
```

Output

```
10
```

```
20
30
```

2.Enumeration interface

It is one of the predefined interface and whose object is always used for retrieving the data from collection framework variable only in forward direction but not in backward direction. Like Iterator interface object, Enumeration Interface object is pointing just before the first element of collection framework variable.

The functionality of Enumeration is more or less similar to Iterator Interface but Enumeration Interface object belongs to synchronized and Iterator Interface object belong to non-synchronized.

Methods:

- **public boolean hasMoreElements():** Return true if Enumeration contains more elements otherwise returns false.
- **public object nextElement():** Returns the next elements of Enumeration.

Syntax

```
Vector v=new Vector();
Enumeration e=v.elements();
```

Example

```
 import java.util.Arraylist;
class EnumerationDemo
 {
 public static void main(String args[])
 {
 Vector<Integer> v=new vector<Integer>();  // creating Vector
 v.add(10);
 v.add(20);
 v.add(30);
 Enumeration e=v.elements();
 while(e.hasMoreElements())
 {
 System.out.println(e.nextElements());
   }  }  }
```

Output

```
10
20
30
```

Collection Classes

1.ArrayList in Java

ArrayList is a replacement of vector class, It is a new class used to store multiple objects.

In ArrayList the data is organizing in the form of cells. Cell values are storing in heap memory and cell address are storing in associative memory.

Creating ArrayList is nothing but creating an object of ArrayList class.

Syntax

ArrayList al=**new** ArrayList();

Difference Between Vector and ArrayList

	Vector	**ArrayList**
1	Vector is legacy Collection Framework (old class).	ArrayList is new Collection Framework.
2	Vector is Synchronized by default.	ArrayList is not Synchronized.
3	For retrieving elements from Vector class can be use foreach loop, iterator, listiterator and enumeration.	For retrieving elements from ArrayList class can be use foreach loop, iterator and listiterator.

Example:

```
import java.util.Arraylist;
class DemoArraylist
{
public static void main(String args[])
{
ArrayList<Integer> al=new ArrayList<Integer>();  // creating arraylist
al.add(10);
al.add(20);
al.add(30);
Iterator itr=al.iterator();  // getting Iterator from arraylist to traverse elements
while(itr.hasNext())
{
System.out.println(itr.next());
  }
 }
}
```

Output

```
10
20
30
```

2.Stack in Java

Stack is one of the sub-class of Vector class so that all the methods of Vector are inherited into Stack.The concept of Stack of Data Structure is implemented in java and develop a pre-defined class called Stack.Stack work on Last in First out (LIFO) manner.

Syntax

Stack s=**new** Stack();

Methods of Stack:

- **public boolean empty():** is used for returns true provided Stack is empty.It returns false in case of Stack is non-empty.
- **public void push (Object):** is used for inserting the elements into the Stack.
- **public Object pop():** is used for removing Top Most elements from the Stack.
- **public Object peek():** is used for retrieving Top Most element from the Stack.
- **public int search(Object):** is used for searching an element in the Stack.If the element is found then it returns Stack relative position of that element otherwise it returns -1, -1 indicates search is unsuccessful and element is not found.

Example

```
import java.util.*;
class StackDemo
{
public static void main(String args[])
{
Stack s=new Stack();
System.out.println("content of s="+s);
System.out.println("size of s="+s.size());
System.out.println("Is empty?="s.empty());
s.push(10);
s.push(20);
s.push(30);
s.push(40);
System.out.println("content of s="+s);
System.out.println("size of s="+s.size());
System.out.println("Is s empty ?=s.empty()");
//remove the top most element
System.out.println("delete element="+s.pop());
System.out.println("content of s after pop="+s);
System.out.println("content of s after peek="+s);
//Search the element 10 and 100
int srp=s.search(10);
 System.out.println("stack relative pos.of 10 is="+srp);
}
}
```

3.Vector in Java

It is one of the Legacy Collection framework class. Vector class object organizes the data in the form of cells. Creating a Vector is nothing but creating an object of Vector class.

Syntax

Vector v=**new** Vector();

Methods:

- **public int capacity():** are used for find the capacity of Vector.
- **public int size():** are used for find size of Vector class object.
- **public void addElement(Object):** are used for adding the elements to the Vector one-by-one.
- **public void addElementAt(int.Object):** are used for adding the elements to the Vector at specific existing position.
- **public Object removeElementAt(int):** are used for removing the elements of Vector on the basics of the position.
- **public void removeElement(Object):** are used for removing the elements of Vector on the basics of content.
- **public Enumeration elements():** are used for extracting all the elements of Vector class object

Example

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

```
class VectorDemo
{
 public static void main(String [] args)
 {
 Vector v=new Vector();
// Add data to v
v.addElement("Deo");
v.addElement("Smith");
v.addElement("Karter");
v.addElement("Porter");
// Extract the data
  Enumeration e=v.elements();
  while(e.hasMoreElements()){
  System.out.println(e.nextElement());
  }
 }
}
```

Output

```
Deo
Smith
Karter
Porter
```

4.HashTable in Java

HashTable is **Implementer** class of Map interface and extends Dictionary class. HashTable does not allows null key and null values, these elements will be stored in a random order.

Note: HashTable class also contains same methods like HashMap.

Example

```
import java.util.*;
class HashTableDemo
{
public static void main(String args[])
{
  HashTable<Integer,String> ht=new HashTable<Integer,String>();
  ht.put(1,"Deo");
  ht.put(2,"zen");
```

```
        ht.put(3,"porter");
        ht.put(4,"piter");
        System.out.println(ht);  }  }
```

Output

```
    [1= porter, 2=zen, 3= Deo,4=piter]
```

Java HashSet

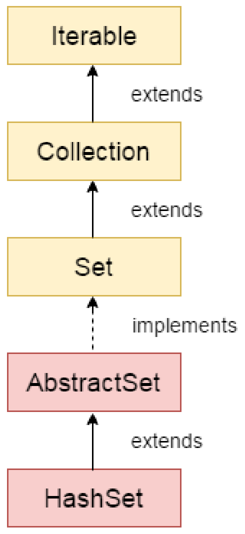
Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called **hashing.**
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.
- The initial default capacity of HashSet is 16, and the load factor is 0.75.

Difference between List and Set

A list can contain duplicate elements whereas Set contains unique elements only.



Methods of Java HashSet class

Various methods of Java HashSet class are as follows:

SN	Modifier & Type	Method	Description
1)	boolean	add(E e)	It is used to add the specified element to this set if it is not already present.
2)	void	clear()	It is used to remove all of the elements from the set.
3)	object	clone()	It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned.
4)	boolean	contains(Object o)	It is used to return true if this set contains the specified element.
5)	boolean	isEmpty()	It is used to return true if this set contains no elements.
6)	Iterator<E>	iterator()	It is used to return an iterator over the elements in this set.
7)	boolean	remove(Object o)	It is used to remove the specified element from this set if it is present.
8)	int	size()	It is used to return the number of elements in the set.
9)	Spliterator<E>	spliterator()	It is used to create a late-binding and fail-fast Spliterator over the elements in the set.

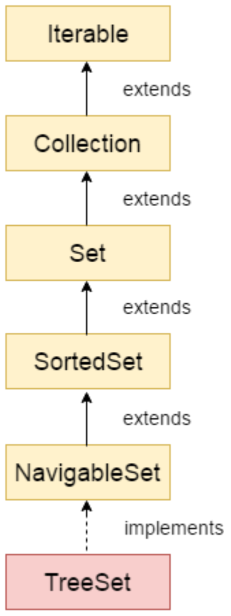
Java HashSet Example

Let's see a simple example of HashSet. Notice, the elements iterate in an unordered collection.

```
import java.util.*;
class HashSet1{
 public static void main(String args[]){
  //Creating HashSet and adding elements
    HashSet<String> set=new HashSet();
           set.add("One");
            set.add("Two");
           set.add("Three");
            set.add("Four");
           set.add("Five");
            Iterator<String> i=set.iterator();
           while(i.hasNext())
            {
           System.out.println(i.next());
            }
 }
}
```

```
Five
One
Four
Two
Three
```

Java TreeSet class



Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

The important points about Java TreeSet class are:

- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class access and retrieval times are quiet fast.
- Java TreeSet class doesn't allow null element.
- Java TreeSet class is non synchronized.
- Java TreeSet class maintains ascending order.

Hierarchy of TreeSet class

As shown in the above diagram, Java TreeSet class implements the NavigableSet interface. The NavigableSet interface extends SortedSet, Set, Collection and Iterable interfaces in hierarchical order.

Java TreeSet Example 1:

```
import java.util.*;
class TreeSet1{
 public static void main(String args[]){
  //Creating and adding elements
  TreeSet<String> al=new TreeSet<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");
  //Traversing elements
  Iterator<String> itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

Java TreeSet Example 3:

```
import java.util.*;
class TreeSet3{
 public static void main(String args[]){
 TreeSet<Integer> set=new TreeSet<Integer>();
         set.add(24);
         set.add(66);
         set.add(12);
         set.add(15);
         System.out.println("Highest Value: "+set.pollFirst());
         System.out.println("Lowest Value: "+set.pollLast());
 }
}
```

Output:

```
Highest Value: 12
Lowest Value: 66
```

Java TreeSet Example 4:

```
import java.util.*;
class TreeSet4{
 public static void main(String args[]){
  TreeSet<String> set=new TreeSet<String>();
         set.add("A");
         set.add("B");
         set.add("C");
         set.add("D");
         set.add("E");
         System.out.println("Initial Set: "+set);

         System.out.println("Reverse Set: "+set.descendingSet());

         System.out.println("Head Set: "+set.headSet("C", true));

         System.out.println("SubSet: "+set.subSet("A", false, "E", true));

         System.out.println("TailSet: "+set.tailSet("C", false));
 }
}
```

Output:

```
Initial Set: [A, B, C, D, E]
Reverse Set: [E, D, C, B, A]
Head Set: [A, B, C]
SubSet: [B, C, D, E]
TailSet: [D, E]
```

Java Calendar Class

Java Calendar class is an abstract class that provides methods for converting date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc. It inherits Object class and implements the Comparable interface.

public abstract class Calendar **extends** Object

implements Serializable, Cloneable, Comparable<Calendar>

Method	**Description**
abstract void add(int field, int amount)	It is used to add or subtract the specified amount of time to the given calendar field, based on the calendar's rules.
int get(int field)	It is used to return the value of the given calendar field.
static Calendar getInstance()	It is used to get a calendar using the default time zone and locale.
abstract int getMaximum(int field)	It is used to return the maximum value for the given calendar field of this Calendar instance.
abstract int getMinimum(int field)	It is used to return the minimum value for the given calendar field of this Calendar instance.
void set(int field, int value)	It is used to set the given calendar field to the given value.
void setTime(Date date)	It is used to set this Calendar's time with the given Date.
Date getTime()	It is used to return a Date object representing this Calendar's time value.

Example

```
import java.util.Calendar;
 public class CalendarExample {
 public static void main(String[] args) {
   Calendar calendar = Calendar.getInstance();
   System.out.println("The current date is : " + calendar.getTime());
   calendar.add(Calendar.DATE, -15);
   System.out.println("15 days ago: " + calendar.getTime());
   calendar.add(Calendar.MONTH, 4);
   System.out.println("4 months later: " + calendar.getTime());
   calendar.add(Calendar.YEAR, 2);
```

```
    System.out.println("2 years later: " + calendar.getTime());
    }
  }
```

Output:

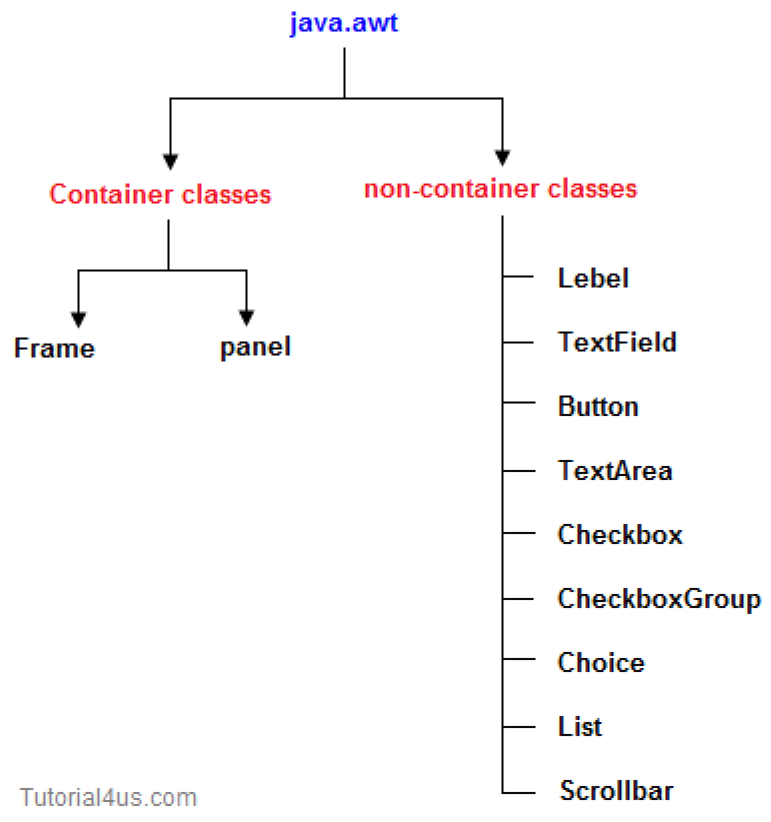
```
The current date is : Thu Jan 19 18:47:02 IST 2017
15 days ago: Wed Jan 04 18:47:02 IST 2017
4 months later: Thu May 04 18:47:02 IST 2017
2 years later: Sat May 04 18:47:02 IST 2019
```

UNIT-V

GUI Programming with Java

AWT stands for **abstract window toolkit**. If any user interact with java program through a graphical window known as **GUI**.
In core java GUI can be design using some predefined classes. All these classes are defined in **java.awt** package.

AWT Class Hierarchy



Container classes:

These are the predefined classes in java.awt package which can be used **to display all non-container classes** to the end user in the frame of window. container classes are; frame, panel.

Non-Container classes:

These are the predefined classes used **to design the input field** so that end user can provide input value to communicate with java program, these are also treated as GUI component. non-container classes are; Label, Button, List etc...

Container Classes

Awt Frame

```
Frame f=new Frame();
```

Mostly used methods

setTitle()

It is used to display user defined message on title bar.

```
Frame f=new Frame();
f.setTitle("myframe");
```

setBackground()

It is used to set background or image of frame.

```
Frame f=new Frame();
f.setBackground(Color.red);
```

setForground()

It is used to set the foreground text color.

```
Frame f=new Frame();
f.setForground(Color.red);
```

setSize()

It is used to set the width and height for frame.

```
Frame f=new Frame();
f.setSize(400,`);
```

setVisible()

It is used to make the frame as visible to end user.

```
Frame f=new Frame();
```

```
f.setVisible(true);
```

Note: You can write setVisible(true) or setVisible(false), if it is true then it visible otherwise not visible.

setLayout()
It is used to set any layout to the frame. You can also set null layout it means no any layout apply on frame.

```
Frame f=new Frame();
f.setLayout(new FlowLayout());
```

Note: Layout is a logical container used to arrange the gui components in a specific order

add()
It is used to add non-container components (Button, List) to the frame.

```
Frame f=new Frame();
Button b=new Button("Click");
f.add(b);
```

Explanation: In above code we add button on frame using f.add(b), here b is the object of Button class..

AWT Controls:

Following is the list of commonly used controls while designed GUI using AWT.

Sr. No.	Control & Description
1	Label A Label object is a component for placing text in a container.
2	Button This class creates a labeled button.

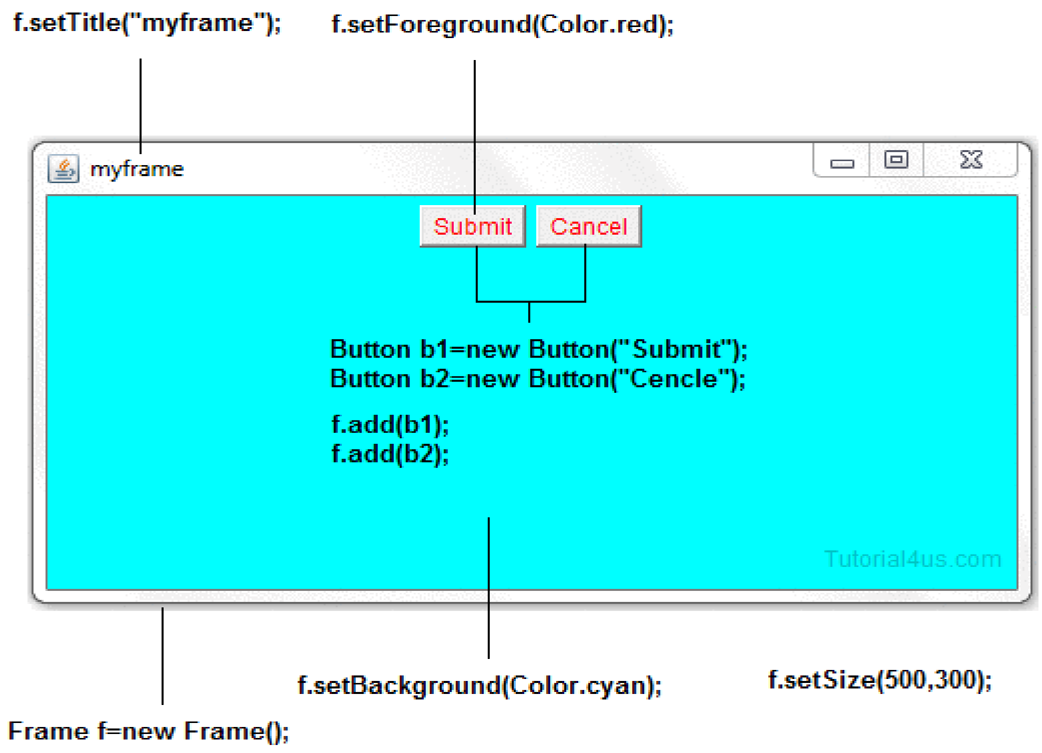
3	Check Box A check box is a graphical component that can be in either an **on** (true) or **off** (false) state.
4	Check Box Group The CheckboxGroup class is used to group the set of checkbox.
5	List The List component presents the user with a scrolling list of text items.
6	Text Field A TextField object is a text component that allows for the editing of a single line of text.
7	Text Area A TextArea object is a text component that allows for the editing of a multiple lines of text.
8	Choice A Choice control is used to show pop up menu of choices. Selected choice is shown on the top of the menu.
12	Dialog A Dialog control represents a top-level window with a title and a border used to take some form of input from the user.

Awt Button Example:

```
import java.awt.*;
class FrameDemo {
public static void main(String[] args)
{
Frame f=new Frame();
f.setTitle("myframe");
```

```
        f.setBackground(Color.cyan);
        f.setForeground(Color.red);
        f.setLayout(new FlowLayout());
        Button b1=new Button("Submit");
        Button b2=new Button("Cancel");
            f.add(b1);f.add(b2);
            f.setSize(500,300);
            f.setVisible(true);
            } }
```

Output


f.setTitle("myframe");
f.setForeground(Color.red);
myframe
Submit
Cancel
Button b1=new Button("Submit");
Button b2=new Button("Cencle");
f.add(b1);
f.add(b2);
Tutorial4us.com
f.setBackground(Color.cyan);
f.setSize(500,300);
Frame f=new Frame();

Java AWT TextField

The object of a **TextField** class is a text component that allows a user to enter a single line text and edit it. It inherits **TextComponent** class, which further inherits **Component** class.

TextFieldExample1.java

```
// importing AWT class
import java.awt.*;
public class TextFieldExample1 {
    // main method
    public static void main(String args[]) {
    // creating a frame
    Frame f = new Frame("TextField Example");

    // creating objects of textfield
    TextField t1, t2;
    // instantiating the textfield objects
    // setting the location of those objects in the frame
    t1 = new TextField("Welcome to Java.");
    t1.setBounds(50, 100, 200, 30);
    t2 = new TextField("AWT Tutorial");
    t2.setBounds(50, 150, 200, 30);
    // adding the components to frame
    f.add(t1);
    f.add(t2);
    // setting size, layout and visibility of frame
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
}
```

Java AWT TextArea

The object of a TextArea class is a multiline region that displays text. It allows the editing of multiple line text. It inherits TextComponent class.

TextAreaExample .java

```
//importing AWT class
import java.awt.*;
public class TextAreaExample
{
// constructor to initialize
    TextAreaExample() {
// creating a frame
        Frame f = new Frame();
// creating a text area
            TextArea area = new TextArea("Welcome to java");
// setting location of text area in frame
        area.setBounds(10, 30, 300, 300);
// adding text area to frame
        f.add(area);
// setting size, layout and visibility of frame
        f.setSize(400, 400);
        f.setLayout(null);
        f.setVisible(true);
    }
// main method
public static void main(String args[])
{
  new TextAreaExample();
}
}
```

Java AWT Checkbox

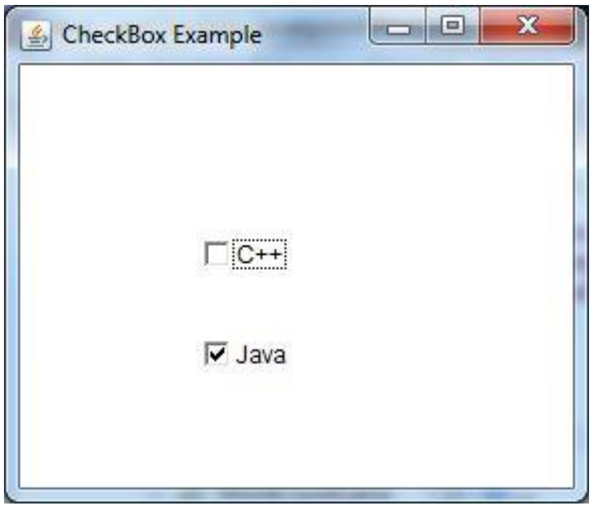
The Checkbox class is used to create a checkbox. It is used to turn an option on (true) or off (false). Clicking on a Checkbox changes its state from "on" to "off" or from "off" to "on".

CheckboxExample1.java

```
// importing AWT class
```

```
import java.awt.*;
public class CheckboxExample1
{
// constructor to initialize
  CheckboxExample1() {
// creating the frame with the title
   Frame f = new Frame("Checkbox Example");
// creating the checkboxes
    Checkbox checkbox1 = new Checkbox("C++");
    checkbox1.setBounds(100, 100,  50, 50);
    Checkbox checkbox2 = new Checkbox("Java", true);
// setting location of checkbox in frame
checkbox2.setBounds(100, 150,  50, 50);
// adding checkboxes to frame
    f.add(checkbox1);
    f.add(checkbox2);

// setting size, layout and visibility of frame
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
  }
// main method
public static void main (String args[])
{
  new CheckboxExample1();
}
}
```

Output:



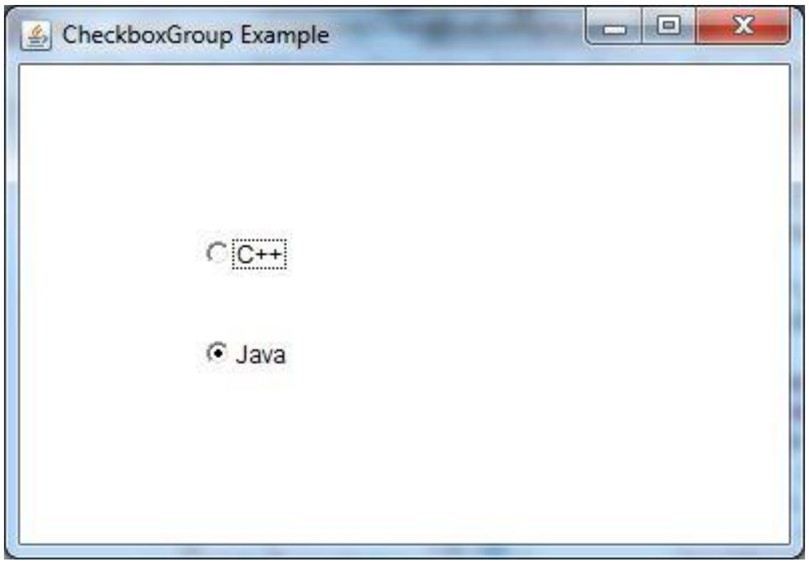
Java AWT CheckboxGroup

The object of CheckboxGroup class is used to group together a set of Checkbox. At a time only one check box button is allowed to be in "on" state and remaining check box button in "off" state. It inherits the object class.

Java AWT CheckboxGroup Example

```
import java.awt.*;

class AWTC

{

  Frame f=new Frame("CHECK BOX");

  CheckboxGroup cbg;

  Checkbox cb1,cb2,cb3,cb4;

  AWTC()

  {

    cbg=new CheckboxGroup();

     f.setSize(400,300);

     cb1=new Checkbox("C",cbg,true);
     cb2=new Checkbox("Java",cbg,false);

     cb3=new Checkbox("Python",cbg,false);

     cb4=new Checkbox(".Net",cbg,false);

     f.setLayout(new FlowLayout());

     f.add(cb1);

     f.add(cb2);

     f.add(cb3);

     f.add(cb4);

     f.setVisible(true);

 }

}

class AWTCD

{

 public static void main(String arg[])

 {

   new AWTC();

 }

}
```

Output:



Java AWT Choice

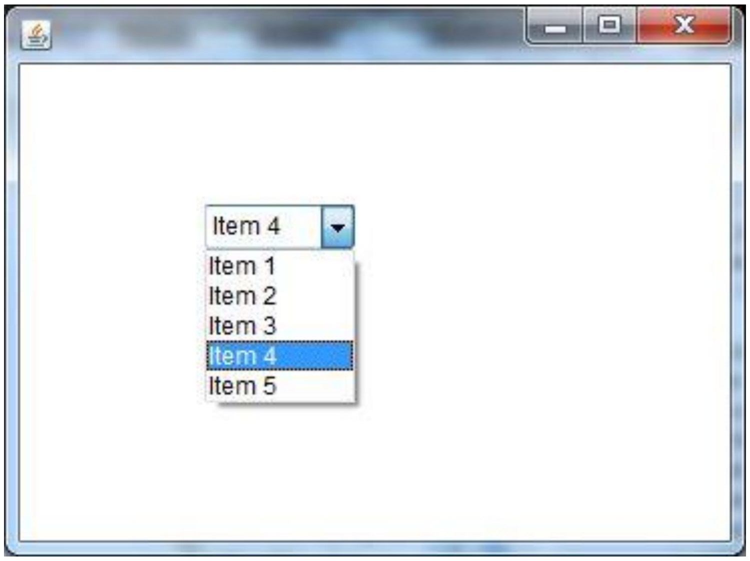
The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits Component class.

ChoiceExample1.java

```
// importing awt class
import java.awt.*;
public class ChoiceExample1 {

     // class constructor
    ChoiceExample1() {

    // creating a frame
    Frame f = new Frame();

    // creating a choice component
    Choice c = new Choice();

    // setting the bounds of choice menu
    c.setBounds(100, 100, 75, 75);
```

```
        // adding items to the choice menu
        c.add("Item 1");
        c.add("Item 2");
        c.add("Item 3");
        c.add("Item 4");
        c.add("Item 5");

        // adding choice menu to frame
        f.add(c);

        // setting size, layout and visibility of frame
        f.setSize(400, 400);
        f.setLayout(null);
        f.setVisible(true);
      }

    // main method
    public static void main(String args[])
    {
      new ChoiceExample1();
    }
    }
```

Output:



Java AWT List

The object of List class represents a list of text items. With the help of the List class, user can choose either one item or multiple items. It inherits the Component class.

Java AWT List Example

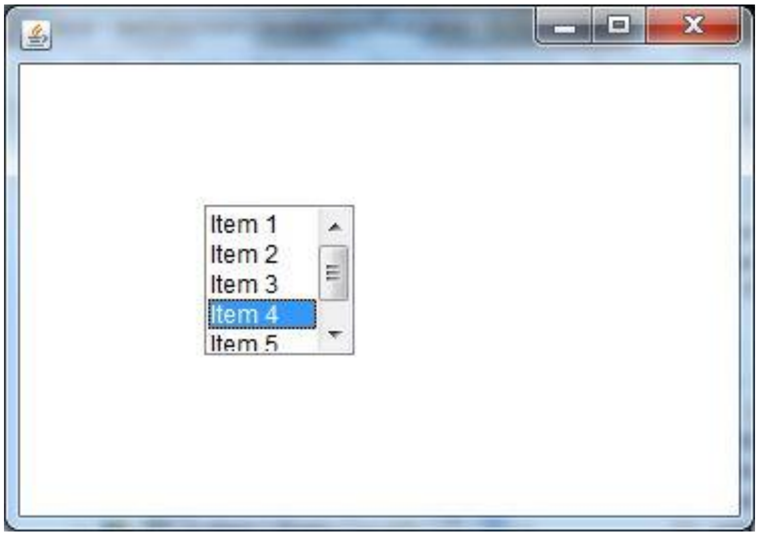
In the following example, we are creating a List component with 5 rows and adding it into the Frame.

ListExample1.java

```
// importing awt class
import java.awt.*;

public class ListExample1
{
    // class constructor
     ListExample1() {
     // creating the frame
         Frame f = new Frame();
       // creating the list of 5 rows
         List l1 = new List(5);
```

```
        // setting the position of list component
        l1.setBounds(100, 100, 75, 75);

        // adding list items into the list
         l1.add("Item 1");
        l1.add("Item 2");
         l1.add("Item 3");
        l1.add("Item 4");
         l1.add("Item 5");

         // adding the list to frame
        f.add(l1);

        // setting size, layout and visibility of frame
         f.setSize(400, 400);
        f.setLayout(null);
         f.setVisible(true);
      }

  // main method
  public static void main(String args[])
  {
     new ListExample1();
  }
  }
```

Output:



Awt Panel

It is a predefined class used to provide a logical container to hold various GUI component. Panel always should exist as a part of frame.

Note: Frame is always visible to end user where as panel is not visible to end user.

Panel is a derived class of container class so you can use all the methods which is used in frame.

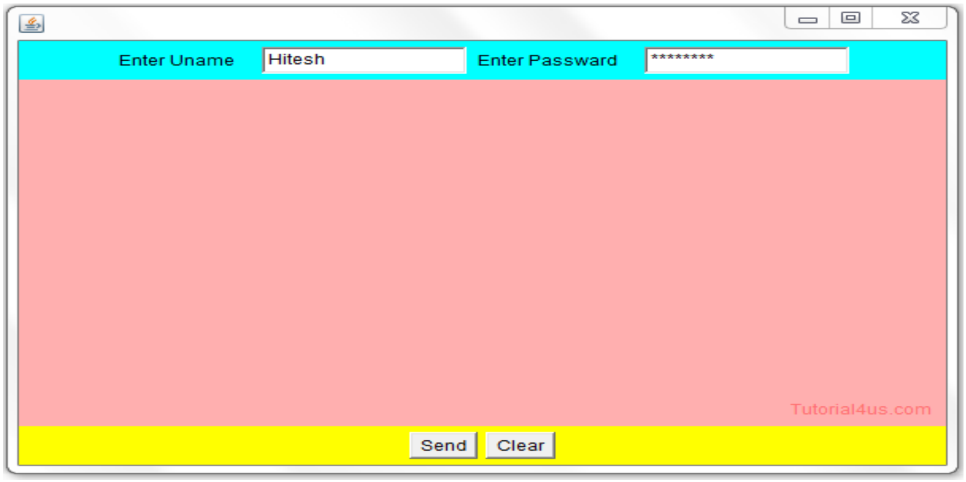
Syntax

```
Panel p=new Panel();
p.setBackground(Color.red);
p.setSize(400,300);
```

Example frame and panel

```
import java.awt.*;
class PanelFrame {
PanelFrame(){
Frame f=new Frame();
f.setSize(600,400);
f.setBackground(Color.pink);
f.setLayout(new BorderLayout());
Panel p1=new Panel();
p1.setBackground(Color.cyan);
Label l1 =new Label("Enter Uname");
TextField tf1=new TextField(15);
Label l2=new Label("Enter Passward");
TextField tf2=new TextField(15);
p1.add(l1);p1.add(tf1);
p1.add(l2);p1.add(tf2);
f.add("North",p1);
Panel p2=new Panel();
p2.setBackground(Color.yellow);
Button b1=new Button("Send");
Button b2=new Button("Clear");
p2.add(b1);
p2.add(b2);
f.add("South",p2);
f.setVisible(true);}
public static void main(String[] args)
{
PanelFrame pf=new PanelFrame();
} }
```

Output:



Awt Layout Management

Layout is a logical container used to arrange the GUI component in proper order within the frame, in java.awt package some of the layout is existing an predefined classes as shown below;

1.FlowLayout
2.BoarderLayout
3.GridLayout

1.FlowLayout

This layout is used to arrange the GUI components in a sequential flow (that means one after another in horizontal way)

You can also set flow layout of components like flow from left, flow from right.

FlowLayout Left

```
Frame f=new Frame();
```

```
f.setLayout(new FlowLayout(FlowLayout.LEFT));
```

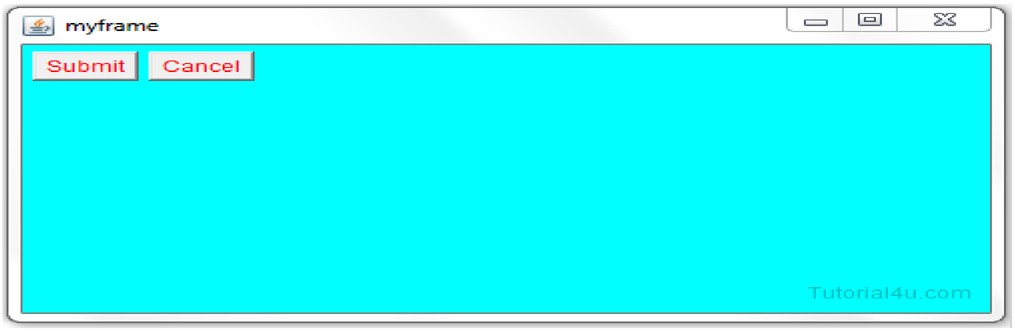
FlowLayout Right

```
Frame f=new Frame();
f.setLayout(new FlowLayout(FlowLayout.RIGHT));
```

Example of FlowLayout

```
import java.awt.*;

class FlowLayoutDemo
{
        public static void main(String[] args)
        {
                Frame f=new Frame();
                f.setTitle("myframe");
                f.setBackground(Color.cyan);
                f.setForeground(Color.red);
                f.setLayout(new FlowLayout(FlowLayout.LEFT));
                Button b1=new Button("Submit");
                Button b2=new Button("Cancel");
                f.add(b1);
                f.add(b2);
                f.setSize(500,300);
                f.setVisible(true);} }
```

Output:



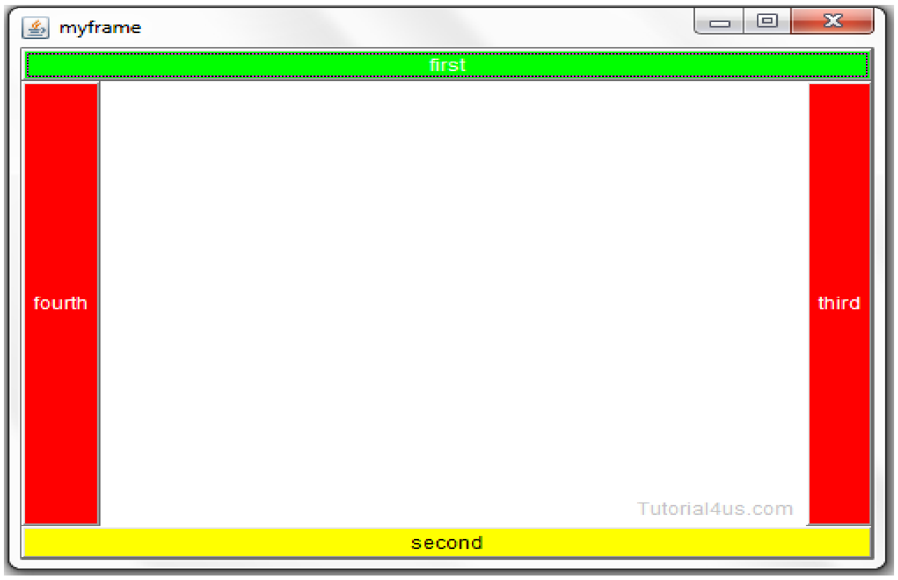
2.BoarderLayout

This layout is used to arrange the GUI components in S directions of frame as shown below.

Example

```
import java.awt.*;
class BorderDemo{
        public static void main(String[] args) {
                Frame f=new Frame();
                f.setTitle("myframe");
                f.setSize(500,400);
                f.setBackground(Color.white);
                f.setLayout(new BorderLayout());
                Button b1=new Button("first");
                b1.setBackground(Color.green);
                b1.setForeground(Color.white);
                Button b2=new Button("second");
                b2.setBackground(Color.yellow);
                b2.setForeground(Color.black);
               Button b3=new Button("third");
                b3.setBackground(Color.red);
                b3.setForeground(Color.white);
                Button b4=new Button("fourth");
                b4.setBackground(Color.red);
                b4.setForeground(Color.white);
               Button b5=new Button("center");
                b5.setBackground(Color.cyan);
                b5.setForeground(Color.white);
               f.add("North",b1);
                f.add("South",b2);
                f.add("East",b3);
                f.add("West",b4);
                f.add(b5);
                f.setVisible(true);} }
```

Output



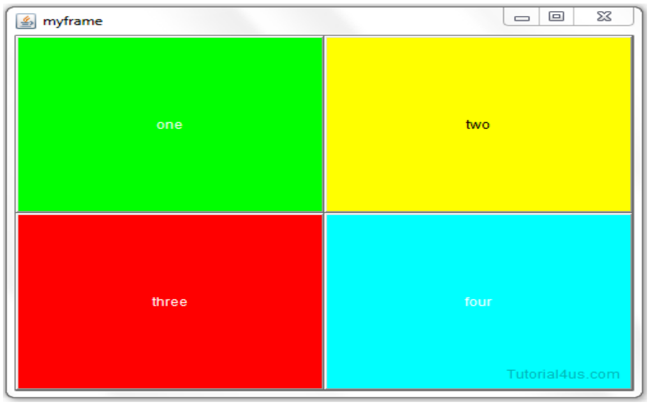
3.GridLayout

This layout is used to arrange the GUI components in the table format.

Example of GridLayout

```
 import java.awt.*;
 class Grid extends Frame
 {
            Grid()
            {
                    Frame f=new Frame();
                    f.setTitle("myframe");
                    f.setSize(500,400);
                    f.setBackground(Color.white);
                    f.setLayout(new GridLayout(2,2));
                    Button b1=new Button("one");
                    b1.setBackground(Color.green);
                    b1.setForeground(Color.white);
                  Button b2=new Button("two");
                    b2.setBackground(Color.yellow);
                    b2.setForeground(Color.black);
                  Button b3=new Button("three");
                    b3.setBackground(Color.red);
                    b3.setForeground(Color.white);
                  Button b4=new Button("four");
                    b4.setBackground(Color.cyan);
                    b4.setForeground(Color.white);
                  f.add("one",b1);
                    f.add("two",b2);
                    f.add("three",b3);
                    f.add("four",b4);
                    f.setVisible(true);}};
class GridDemo
{
        public static void main(String [] args)
        {
                Grid g1=new Grid();
        } }
```

Output



Introduction to Swing

Swing is *used to create window-based applications*. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.
Unlike AWT, Java Swing provides platform-independent and lightweight components.

The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu etc.

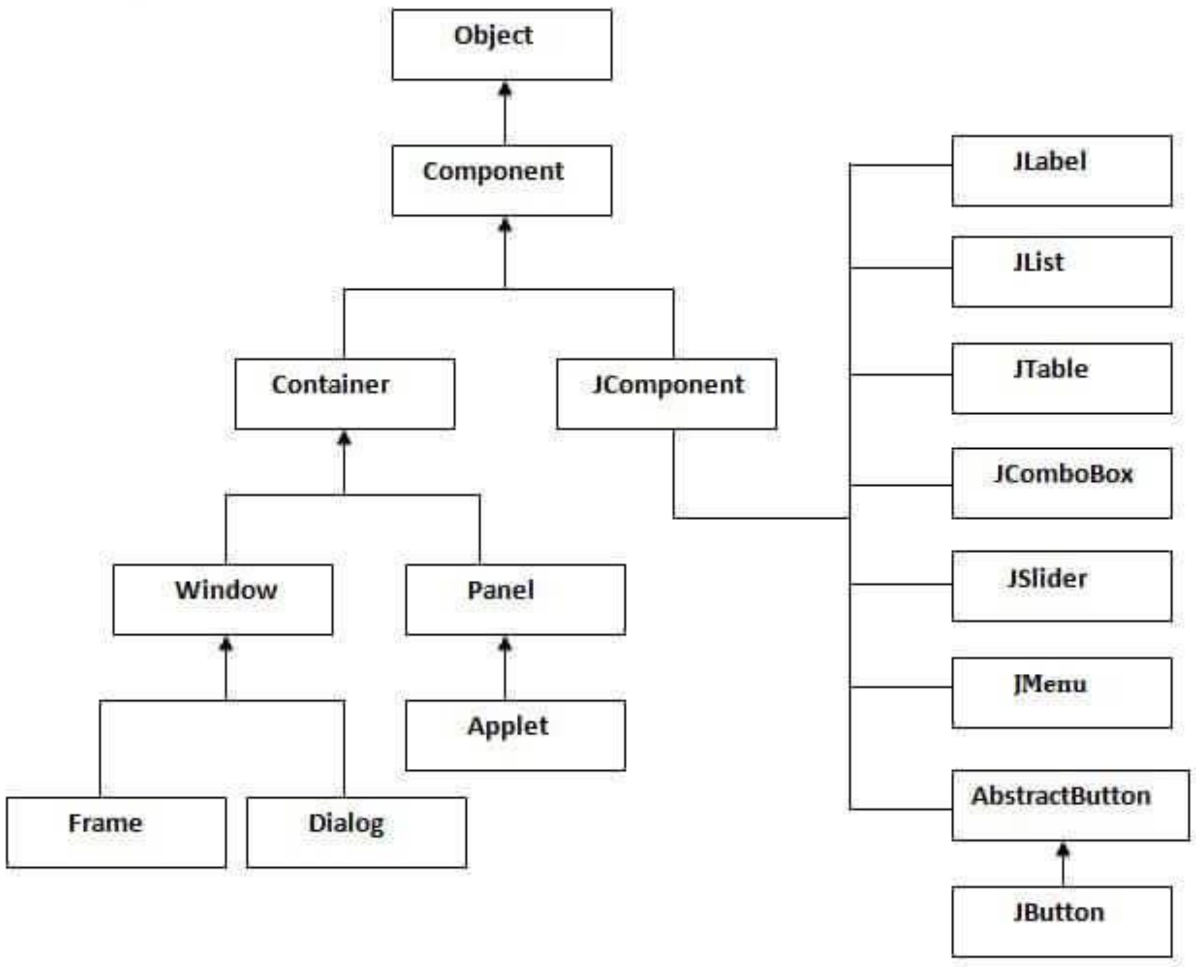
Difference between AWT and Swing

There are many differences between java awt and swing that are given below.

No.	Java AWT	Java Swing
1	AWT components are **platform-dependent**.	Java swing components are**platform-independent**.
2	AWT components are **heavyweight**.	Swing components are **lightweight**.
3	AWT **doesn't support pluggable**	Swing **supports pluggable look and feel**.

	look and feel.	
4	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes,colorchooser,tabbedpane etc.
5	AWT **doesn't follows MVC**(Model View Controller).	Swing **follows MVC**.

Hierarchy of Java Swing classes



Commonly used Methods of Component class

The methods of Component class are widely used in java swing that are given below.

Method	**Description**
public void add(Component c)	add a component on another component.
public void setSize(int width,int height)	sets size of the component.
public void setLayout(LayoutManager m)	sets the layout manager for the component.
public void setVisible(boolean b)	sets the visibility of the component. It is by default false.

Containers

JFrame:

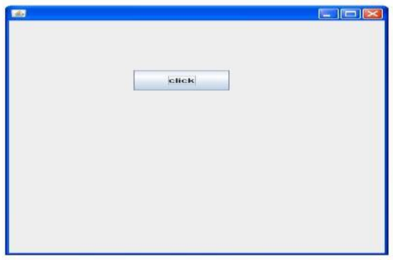
There are two ways to create a frame:

1.By creating the object of JFrame class (association)
2.By extending JFrame class (inheritance)

1.By creating the object of Frame class (association)

```
import javax.swing.*;
public class FirstSwingExample {
public static void main(String[] args) {
JFrame f=new JFrame();
JButton b=new JButton("click");
f.add(b);
f.setSize(400,500);
f.setVisible(true);
}  }
```

Output:



2.By extending Frame class (inheritance)

We can also inherit the JFrame class, so there is no need to create the instance of JFrame class explicitly.

```
import javax.swing.*;
```

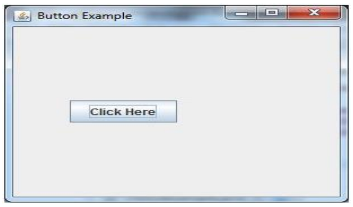
```
public class Simple2 extends JFrame{//inheriting JFrame
JFrame f;
Simple2(){
JButton b=new JButton("click");//create button
add(b);//adding button on frame
setSize(400,500);
setVisible(true);
}
public static void main(String[] args) {
new Simple2();
}}
```

Swing Components

1.JButton

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

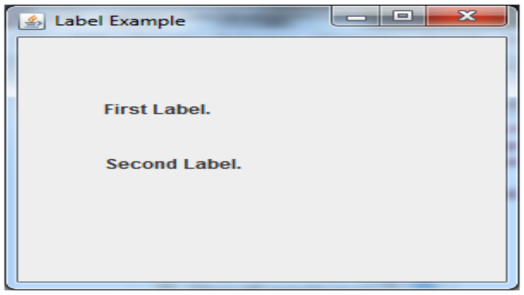
Output:



2. JLabel

```
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.");
    l2=new JLabel("Second Label.");
    f.add(l1); f.add(l2);
    f.setSize(300,300);
f.setVisible(true);
}
}
```

Output:

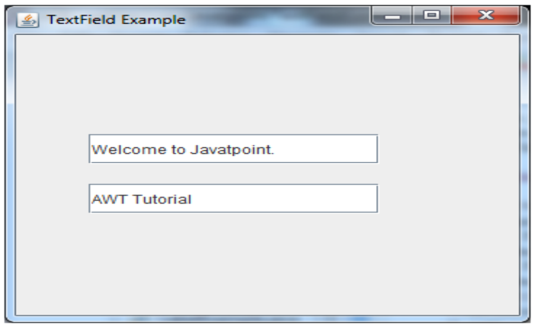


```
First Label.

Second Label.
```

3.JTextField

```
import javax.swing.*;
class TextFieldExample {
public static void main(String args[]){
    JFrame f= new JFrame("TextField Example");
    JTextField t1,t2;
    t1=new JTextField("Welcome to Javatpoint.");
    t2=new JTextField("AWT Tutorial");
    f.add(t1); f.add(t2);
    f.setSize(400,400);
    f.setVisible(true);
    } }
```

Output:



```
Welcome to Javatpoint.

AWT Tutorial
```

Event Handling in AWT

In general you can not perform any operation on dummy GUI screen even any button click or select any item. To perform some operation on these dummy GUI screen you need some predefined classes and interfaces. All these type of classes and interfaces are available in **java.awt.event** package.

Changing the state of an object is known as an **event**.

The process of handling the request in GUI screen is known as **event handling** (event represent an action). It will be changes component to component.

Note: In event handling mechanism event represent an action class and Listener represent an interface. Listener interface always contains abstract methods so here you need to write your own logic.

Source - Event source is an object that generates an event.Source is responsible for providing information of the occurred event to it's handler.

Listener - It is also known as event handler. Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event and then returns.

Event classes and Listener interfaces

Event Classes	**Description**	**Listener Interface**
ActionEvent	generated when button is pressed, menu-item is selected, list-item is double clicked	ActionListener
MouseEvent	generated when mouse is dragged, moved,clicked,pressed or released and also when it enters or exit a component	MouseListener
KeyEvent	generated when input is received from keyboard	KeyListener
ItemEvent	generated when check-box or list item is clicked	ItemListener
TextEvent	generated when value of textarea or textfield is changed	TextListener
MouseWheelEvent	generated when mouse wheel is moved	MouseWheelListener
WindowEvent	generated when window is activated, deactivated, deiconified, iconified, opened or closed	WindowListener
ComponentEvent	generated when component is hidden,moved,resized orset visible	ComponentEventListener
ContainerEvent	generated when component is added or removed from container	ContainerListener
AdjustmentEvent	generated when scroll bar is manipulated	AdjustmentListener
FocusEvent	generated when component gains or loses keyboard focus	FocusListener

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){}

Steps to perform Event Handling

Following steps are required to perform event handling:

- Implement the Listener interface and overrides its methods
- Register the component with the Listener

Steps involved in Event Handling

- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- the method is now get executed and returns.

Syntax to Handle the Event

```
class className implements XXXListener
{
.......
.......
}
addcomponentobject.addXXXListener(this);
.......
// override abstract method of given interface and write proper logic
public void methodName(XXXEvent e)
{
.......
.......
}
.......
}
```

Event Handling for Mouse

For handling event for mouse you need MouseEvent class and MouseListener interface.

GUI Component	Event class	Listener Interface
Mouse	MouseEvent	MouseListener

The Java MouseListener is notified whenever you change the state of mouse. It is notified against MouseEvent. The MouseListener interface is found in java.awt.event package. It has five methods.

Methods of MouseListener interface

The signature of 5 methods found in MouseListener interface are given below:

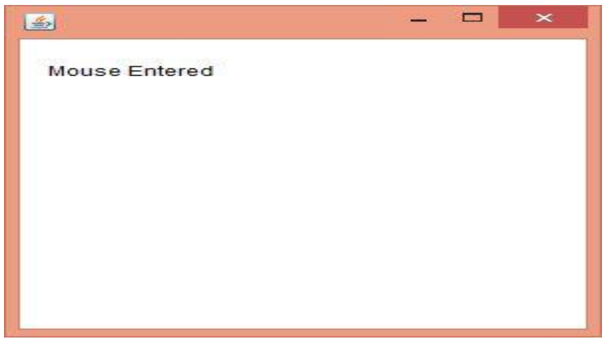
1. **public abstract void** mouseClicked(MouseEvent e);
2. **public abstract void** mouseEntered(MouseEvent e);
3. **public abstract void** mouseExited(MouseEvent e);
4. **public abstract void** mousePressed(MouseEvent e);
5. **public abstract void** mouseReleased(MouseEvent e);

Java MouseListener Example

```
import java.awt.*;
import java.awt.event.*;
public class MouseListenerExample extends Frame implements MouseListener{
   Label l;
   MouseListenerExample(){
      addMouseListener(this);
      l=new Label();
      add(l);
      setSize(300,300);
      setVisible(true);  }
   public void mouseClicked(MouseEvent e) {
      l.setText("Mouse Clicked");   }
   public void mouseEntered(MouseEvent e) {
      l.setText("Mouse Entered");    }
   public void mouseExited(MouseEvent e) {
      l.setText("Mouse Exited");  }
   public void mousePressed(MouseEvent e) {
      l.setText("Mouse Pressed");   }
```

```
    public void mouseReleased(MouseEvent e) {
        l.setText("Mouse Released");     }
public static void main(String[] args) {
    new MouseListenerExample();  }
  }
```

Output:


```
Mouse Entered
```

Event Handling for Button

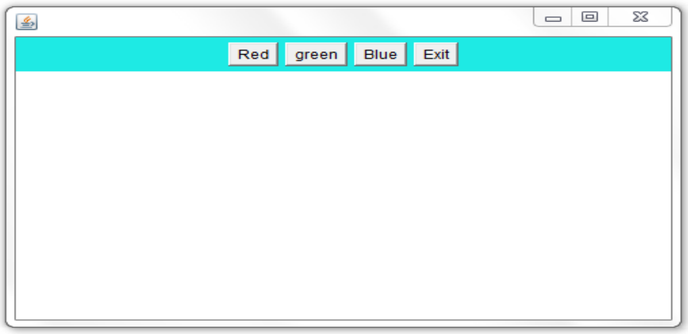
Event handling for button component you need to use ActionEvent class and ActionListener interface. ActionEvent class and ActionListener interface is associated with button component. If any button is clicked operation will be performed by writing the logic in actionPerform().

GUI Component	Event class	Listener Interface	Method (abstract method)
Button	ActionEvent	ActionListener	public void actionPerformed(ActionEvent e)

Example:

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

class A implements ActionListener
{
Frame f;
Button b1,b2,b3,b4;
A()
{
 f=new Frame();
 f.setSize(500,500);
 f.setLayout(new BorderLayout());
 Panel p=new Panel();
 p.setBackground(Color.cyan);
 b1=new Button("Red");
 b2=new Button("green");
 b3=new Button("Blue");
 b4=new Button("Exit");
 p.add(b1);
 p.add(b2);
 p.add(b3);
 p.add(b4);
 f.add("North",p);
 b1.addActionListener(this);
 b2.addActionListener(this);
 b3.addActionListener(this);
 b4.addActionListener(this);
 f.setVisible(true);
}

public void actionPerformed(ActionEvent e)
{
try
{
if(e.getSource().equals(b1))
{
f.setBackground(Color.red);
}
else if(e.getSource().equals(b2))
{
f.setBackground(Color.green);
}
else if(e.getSource().equals(b3))
{
f.setBackground(Color.blue);
}
else if(e.getSource().equals(b4))
{
System.exit(0);
}
}
catch (Exception ec)
{
System.out.println(ec);
}
}
};

class ActionEventEx
{
public static void main(String[] args)
{
A a1=new A();
```

```
    }
    }
```



In above example when you click on button then change background color of frame.

Adapter Classes

In a program, when a listener has many abstract methods to override, it becomes complex for the programmer to override all of them.

For example, for closing a frame, we must override seven abstract methods of WindowListener, but we need only one method of them.

For reducing complexity, Java provides a class known as "adapters" or adapter class. Adapters are abstract classes, that are already being overriden.

java.awt.event Adapter classes

Adapter class	**Listener interface**
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
HierarchyBoundsAdapter	HierarchyBoundsListener

Java WindowAdapter Example

```
import java.awt.*;
import java.awt.event.*;
public class AdapterExample{
    Frame f;
    AdapterExample(){
        f=new Frame("Window Adapter");
        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e) {
                f.dispose();
            }  });
        f.setSize(400,400);
        f.setVisible(true);
    }
public static void main(String[] args) {
    new AdapterExample();
}  }
```

Java Applet

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

Advantage of Applet

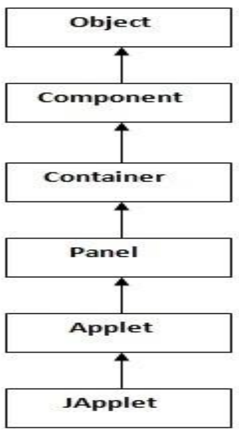
There are many advantages of applet. They are as follows:

- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many plateforms, including Linux, Windows, Mac Os etc.

Drawback of Applet

- Plugin is required at client browser to execute applet.

Hierarchy of Applet



As displayed in the above diagram, Applet class extends Panel. Panel class extends Container which is the subclass of Component.

Lifecycle of Java Applet

1. Applet is initialized.
2. Applet is started.
3. Applet is painted.
4. Applet is stopped.
5. Applet is destroyed.

Lifecycle methods for Applet:

The java.applet.Applet class 4 life cycle methods and java.awt.Component class provides 1 life cycle methods for an applet.

java.applet.Applet class

For creating any applet java.applet.Applet class must be inherited. It provides 4 life cycle methods of applet.

1. **public void init():** is used to initialized the Applet. It is invoked only once.
2. **public void start():** is invoked after the init() method or browser is maximized. It is used to start the Applet.
3. **public void stop():** is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.
4. **public void destroy():** is used to destroy the Applet. It is invoked only once.

java.awt.Component class

The Component class provides 1 life cycle method of applet.

1. **public void paint(Graphics g):** is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc.

How to run an Applet?

There are two ways to run an applet

1. By html file.
2. By appletViewer tool (for testing purpose).

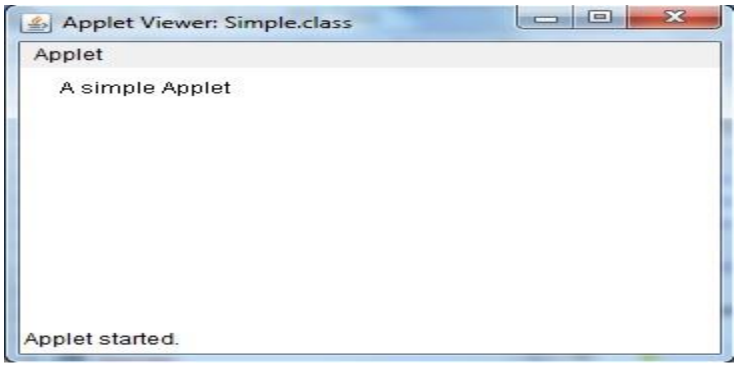
Simple example of Applet by html file:

To execute the applet by html file, create an applet and compile it. After that create an html file and place the applet code in html file. Now click the html file.

```
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet
{
  public void paint(Graphics g)
   {
     g.drawString("A simple Applet",20,20);
   }
}
```

myapplet.html

```
<html>
<body>
<applet code="First.class" width="300" height="300">
</applet>
</body>
</html>
```



A simple Applet

Applet started.

Simple example of Applet by appletviewer tool:

To execute the applet by appletviewer tool, create an applet that contains applet tag in comment and compile it. After that run it by: appletviewer First.java. Now Html file is not required but it is for testing purpose only.

```
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet
{
public void paint(Graphics g)
{
g.drawString("welcome to applet",150,150);
}
}
/*
<applet code="First.class" width="300" height="300">
</applet>
*/
```

To execute the applet by appletviewer tool, write in command prompt:

```
c:\>javac First.java
c:\>appletviewer First.java
```

Parameter in Applet

We can get any information from the HTML file as a parameter. For this purpose, Applet class provides a method named getParameter(). Syntax:

```
public String getParameter(String parameterName)
```

Example of using parameter in Applet:

```
import java.applet.Applet;
import java.awt.Graphics;
public class UseParam extends Applet{
public void paint(Graphics g){
String str=getParameter("msg");
g.drawString(str,50, 50);
}
}
```

myapplet.html

```
<html>
<body>
<applet code="UseParam.class" width="300" height="300">
<param name="msg" value="Welcome to applet">
</applet>
</body>
</html>
```