JAVA PROGRAMMING

Foundations of Java: History of Java, Java Features, Variables, Data Types, Operators, Expressions, Control Statements. Elements of Java - Class, Object, Methods, Constructors and Access Modifiers, Generics, Inner classes, String class and Annotations. OOP Principles: Encapsulation – concept, setter and getter method usage, this keyword. Inheritance - concept, Inheritance Types, super keyword. Polymorphism – concept, Method Overriding usage and Type Casting. Abstraction – concept, abstract keyword and Interface.

HISTORY OF JAVA

- Java is a object oriented programming language.
- Java was created based on C and C++.
- Java uses C syntax and many of the object-oriented features are taken from C++.
- Before Java was invented there were other languages like COBOL, FORTRAN, C, C++, Small Talk, etc.
- These languages had few disadvantages which were corrected in Java.
- Java also innovated many new features to solve the fundamental problems which the previous languages could not solve.
- 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.
- 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.

JAVA BUZZWORDS(JAVA FEATURES)

- Java is the most popular object-oriented programming language.
- Java has many advanced features, a list of key features is known as Java Buzz Words. The

Following list of Buzz Words

1. Simple
2. Secure

3. Portable
4. Object-oriented
5. Robust
6. Architecture-neutral (or) Platform Independent
7. Multi-threaded
8. Interpreted
9. High performance
10. Distributed
11. Dynamic

Simple

- Java programming language is very simple and easy to learn, understand, and code.
- Most of the syntaxes in java follow basic programming language C and object-oriented programming concepts are similar to C++.
- In a java programming language, many complicated features like pointers, operator overloading, structures, unions, etc. have been removed.
- One of the most useful features is the garbage collector it makes java more simple.

Secure

- Java is said to be more secure programming language because it does not have pointers concept.
- java provides a feature "applet" which can be embedded into a web application.
- The applet in java does not allow access to other parts of the computer, which keeps away from harmful programs like viruses and unauthorized access.

Portable

- Portability is one of the core features of java .
- If a program yields the same result on every machine, then that program is called portable.
- Java programs are portable This is the result of java System independence nature.

Object-oriented

- Java is an object oriented programming language.
- This means java programs use objects and classes.

Robust

- Robust means strong.
- Java programs are strong and they don't crash easily like a C or C++ programs There are two reasons
- Java has got excellent inbuilt exception handling features. An exception is an error that occurs at runtime. If an exception occurs, the program terminates suddenly giving rise to problems like loss of data. Overcoming such problem is called exception handling.
- Most of the C and C++ programs crash in the middle because of not allocating sufficient memory or forgetting the memory to be freed in a program. Such problems will not occur in java because the user need not allocate or deallocate the memory in java. Everything will be taken care of by JVM only.

Architecture-neutral (or) Platform Independent

- Java has invented to archive "write once; run anywhere, anytime, forever".
- The java provides JVM (Java Virtual Machine) to archive architectural-neutral or platform independent.
- The JVM allows the java program created using one operating system can be executed on any other operating system.

Multi-threaded

- Java supports multi-threading programming.
- Which allows us to write programs that do multiple operations simultaneously?

Interpreted

- Java programs are compiled to generate byte code.
- This byte code can be downloaded and interpreted by the interpreter in JVM.
- If we take any other language, only an interpreter or a compiler is used to execute the program.
- But in java, we use both compiler and interpreter for the execution.

High performance

- The problem with interpreter inside the JVM is that it is slow.
- Because of Java programs used to run slow.
- To overcome this problem along with the interpreter.
- Java soft people have introduced JIT (Just in Time) compiler, to enhance the speed of execution.
- So now in JVM, both interpreter and JIT compiler work together to run the program.

Distributed

- Information is distributed on various computers on a network.
- Using Java, we can write programs, which capture information and distribute it to the client.
- This is possible because Java can handle the protocols like TCP/IP and UDP.

Dynamic

Java is said to be dynamic because the java byte code may be dynamically updated on a running system and it has a dynamic memory allocation and deallocation (objects and garbage collector).

SUMMARY OF OOP CONCEPTS

- Everything is an object.
- Computation is performed by objects communicating with each other, requesting that other objects perform actions. Objects communicate by sending & receiving messages. A message is a request for an action bundled with whatever arguments may be necessary to complete the task.
- Each object has its own memory, which consists of other objects.
- Every Object is an instance of class. A class simply represents a grouping of similar objects, such as

integers or lists.

- The class is the repository for behavior associated with an object. That is all objects that are instances of same class can perform the same actions.
- Classes are organized into a singly rooted tree structure, called inheritance hierarchy.

OOP CONCEPTS

OOP stands for Object-Oriented Programming. OOP is a programming paradigm in which every program is follows the concept of object. In other words, OOP is a way of writing programs based on the object concept.

The object-oriented programming paradigm has the following core concepts.

- Class
- Object
- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

Class

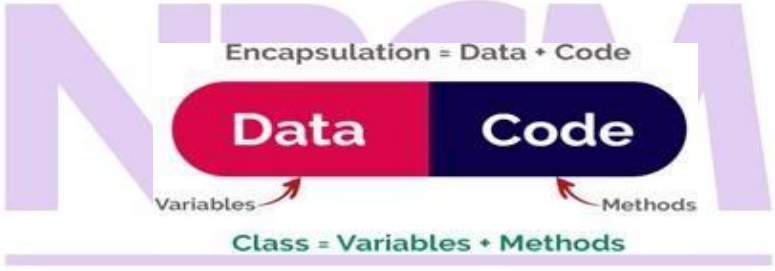
Class is a blue print which is containing only list of variables and methods and no memory is allocatedfor them. A class is a group of objects that has common properties.

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.

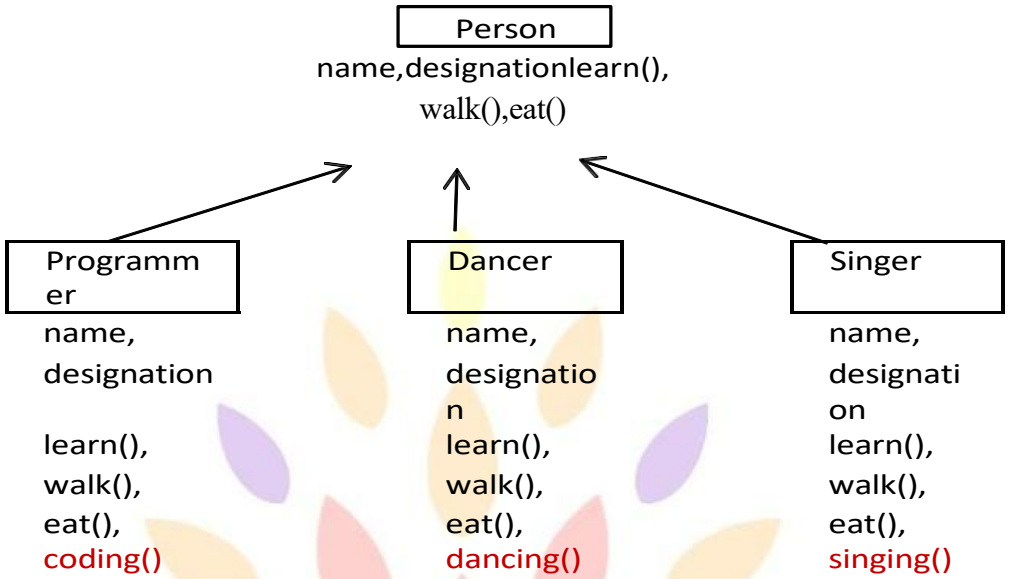
Encapsulation

- Encapsulation is the process of combining data and code into a single unit.
- In OOP, every object is associated with its data and code.
- In programming, data is defined as variables and code is defined as methods.
- The java programming language uses the class concept to implement encapsulation.



Inheritance

- Inheritance is the process of acquiring properties and behaviors from one object to another object or one class to another class.
- In inheritance, we derive a new class from the existing class. Here, the new class acquires the properties and behaviors from the existing class.
- In the inheritance concept, the class which provides properties is called as parent class and the class which recieves the properties is called as child class.



Polymorphism

- Polymorphism is the process of defining same method with different implementation. That means creating multiple methods with different behaviors.
- The java uses method overloading and method overriding to implement polymorphism.
- Method overloading - multiple methods with same name but different parameters.
- Method overriding - multiple methods with same name and same parameters.

Abstraction

- Abstraction is hiding the internal details and showing only essential functionality.
- In the abstraction concept, we do not show the actual implementation to the end user, instead we provide only essential things.
- For example, if we want to drive a car, we does not need to know about the internal functionality like how wheel system works? how brake system works? how music system works? etc.

COPING WITH COMPLEXITY

- Coping with complexity in Java, or any programming language, is an essential skill for software developers.
- As your Java projects grow in size and complexity, maintaining, debugging, and extending yourcode can become challenging.

Here are some strategies to help you cope with complexity in Java:

2. Modularization: Breaking down the code into smaller, self-contained modules, classes, or packages to manage complexity. Each module should have a specific responsibility and interact with others through well-defined interfaces.

3. Design Patterns: Using established design patterns to solve common architectural and design problems. Design patterns provide proven solutions to recurring challenges in software development.

4. Encapsulation: Restricting access to class members using access modifiers (public, private, protected) to hide implementation details and provide a clear API for interacting with the class.

5. Abstraction: Creating abstract classes and interfaces to define contracts that classes must adhere to, making it easier to work with different implementations.

6. Documentation: Writing comprehensive documentation, including code comments and Javadoc, to explain how the code works, its purpose, and how to use it.

7. Testing: Implementing unit tests to ensure that individual components of the code function correctly, which helps identify and prevent bugs.

8. Code Reviews: Collaborating with team members to review and provide feedback on code to catch issues and ensure code quality.

9. Version Control: Using version control systems like Git to manage changes, track history, and collaborate with others effectively.

10. Refactoring: Regularly improving and simplifying the codebase by removing redundancy and improving its structure.

ABSTRACTION MECHANISM

- In Java, abstraction is a fundamental concept in object-oriented programming that allows you to hide complex implementation details while exposing a simplified and well-defined interface.
- Abstraction mechanisms in Java include the use of abstract classes and interfaces.

A WAY OF VIEWING WORLD

- A way of viewing the world is an idea to illustrate the object-oriented programming concept with an example of a real-world situation.
- Let us consider a situation, I am at my office and I wish to get food to my family members who are at my home from a hotel. Because of the distance from my office to home, there is no possibility of getting food from a hotel myself. So, how do we solve the issue?
- To solve the problem, let me call zomato (an agent in food delevery community), tell them the variety and quantity of food and the hotel name from which I wish to delever the food to my family members.

AGENTS AND COMMUNITIES

Let us consider a situation, I am at my office and I wish to get food to my family members who are at my home from a hotel. Because of the distance from my office to home, there is no possibility of getting food from a hotel myself. So, how do we solve the issue?

To solve the problem, let me call zomato (an agent in food delevery community), tell them the variety and quantity of food and the hotel name from which I wish to delever the food to my familymembers. An object-oriented program is structured as a community of interacting agents, called objects. Where each object provides a service (data and methods) that is used by other members of the community.

In our example, the online food delivery system is a community in which the agents are zomato and set of hotels. Each hotel provides a variety of services that can be used by other members like zomato, myself, and my family in the community.

RESPONSIBILITIES

In object-oriented programming, behaviors of an object described in terms of responsibilities. In our

example, my request for action indicates only the desired outcome (food delivered to my

family). The agent (zomato) free to use any technique that solves my problem. By discussing a problem in terms of responsibilities increases the level of abstraction. This enables more independence between the objects in solving complex problems.

MESSAGES & METHODS

To solve my problem, I started with a request to the agent zomato, which led to still more requests among the members of the community until my request has done. Here, the members of a community interact with one another by making requests until the problem has satisfied.

In object-oriented programming, every action is initiated by passing a message to an agent (object), which is responsible for the action. The receiver is the object to whom the message was sent. In response to the message, the receiver performs some method to carry out the request.
Every message may include any additional information as arguments.

In our example, I send a request to zomato with a message that contains food items, the quantity of food, and the hotel details. The receiver uses a method to food get delivered to my home.

VARIABLES

Variable is a name given to a memory location where we can store different values of the same data type during the program execution.
The following are the rules to specify a variable name...

- A variable name may contain letters, digits and underscore symbol
- Variable name should not start with digit.
- Keywords should not be used as variable names.
- Variable name should not contain any special symbols except underscore(_).
- Variable name can be of any length but compiler considers only the first 31 characters of the variable name.

Declaration of Variable
Declaration of a variable tells to the compiler to allocate required amount of memory with specified variable name and allows only specified datatype values into that memory location.
Syntax: datatype variablename; Example : int a;
Syntax : data_type variable_name_1, variable_name_2,...; Example : int a, b;

Initialization of a variable:
Syntax: datatype variablename = value; Example : int a = 10;
Syntax : data_type variable_name_1=value, variable_name_2 = value; Example : int a = 10, b = 20;
SCOPE AND LIFETIME OF A VARIABLE

- In programming, a variable can be declared and defined inside a class, method, or block.
- It defines the scope of the variable i.e. the visibility or accessibility of a variable.
- Variable declared inside a block or method are not visible to outside.
- If we try to do so, we will get a compilation error. Note that the scope of a variable can be nested.
- Lifetime of a variable indicates how long the variable stays alive in the memory.

TYPES OF VARIABLES IT'S SCOPE
There are three types of variables in Java:

1. local variable
2. instance variable
3. static variable

Local Variables

- Variables declared inside the methods or constructors or blocks are called as local variables.
- The scope of local variables is within that particular method or constructor or block in which they have been declared.
- Local variables are allocated memory when the method or constructor or block in which they are declared is invoked and memory is released after that particular method or constructor or block is executed.
- Access modifiers cannot be assigned to local variables.
- It can't be defined by a static keyword.
- Local variables can be accessed directly with their name.

Program

```
class LocalVariables
{
public void show()
{
int a = 10;
System.out.println("Inside show method, a = " + a);
}
public void display()
{
int b = 20;
System.out.println("Inside display method, b = " + b);
//System.out.println("Inside display method, a = " + a); // error
}
public static void main(String args[])
{
LocalVariables obj = new LocalVariables(); obj.show();
obj.display();
}

}
```

Instance Variables:

- Variables declared outside the methods or constructors or blocks but inside the class are called as instance variables.
- The scope of instance variables is inside the class and therefore all methods, constructors and blocks can access them.
- Instance variables are allocated memory during object creation and memory is released during object destruction. If no object is created, then no memory is allocated.
- For each object, a separate copy of instance variable is created.
- Heap memory is allocated for storing instance variables.
- Access modifiers can be assigned to instance variables.
- It is the responsibility of the JVM to assign default value to the instance variables as per the type of Variable.
- Instance variables can be called directly inside the instance area.
- Instance variables cannot be called directly inside the static area and necessarily requires an object reference for calling them.

Program

```
class InstanceVariable
{
int x = 100;
public void show()
{
System.out.println("Inside show method, x = " + x); x = x + 100;
}
public void display()
{
System.out.println("Inside display method, x = " + x);
}
```

```
public static void main(String args[])
{
ClassVariables obj = new ClassVariables(); obj.show();
obj.display();
}
}
```

Static variables

- Static variables are also known as class variable.
- Static variables are declared with the keyword 'static ' .
- A static variable is a variable whose single copy in memory is shared by all the objects, any modification to it will also effect other objects.
- Static keyword in java is used for memory management, i.e it saves memory.
- Static variables gets memory only once in the class area at the time of class loading.
- Static variables can be invoked without the need for creating an instance of a class.
- Static variables contain values by default. For integers, the default value is 0. For Booleans, it is false. And for object references, it is null.

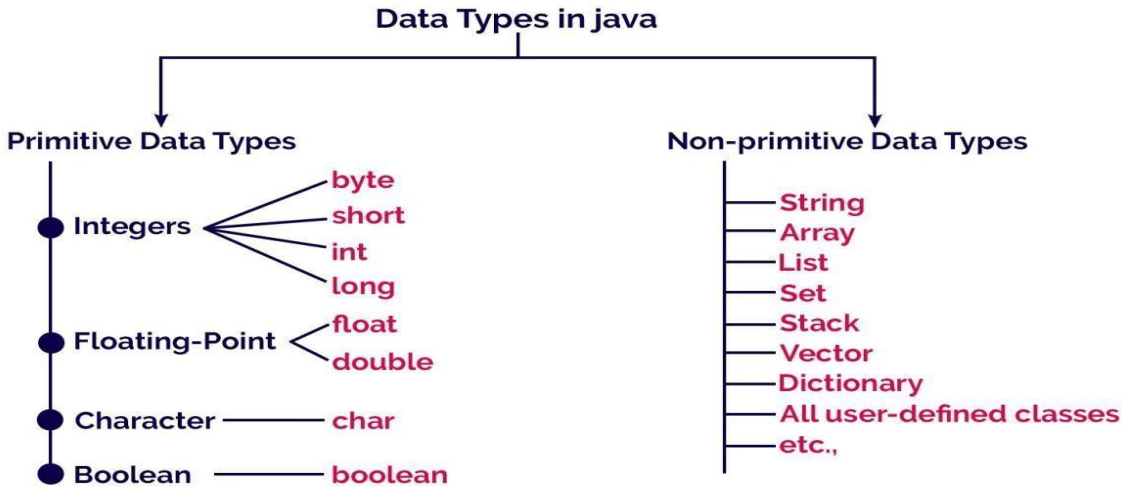
Syntax: static datatype variable name; Example: static int x=100;
Syntax: classname.variablename;
Example

```
class Employee
{
static int empid=500; static void emp1()
{




empid++;
System.out.println("Employee id:"+empid);
}
}
class Sample
{
public static void main(String args[])
{
Employee.emp1(); Employee.emp1(); Employee.emp1(); Employee.emp1(); Employee.emp1(); Employee.emp1();
}
}
```

DATA TYPES

Java programming language has a rich set of data types. The data type is a category of data stored in variables. In java, data types are classified into two types and they are as follows.

- Primitive Data Types
- Non-primitive Data Types



Primitive Data Types

The primitive data types are built-in data types and they specify the type of value stored in a variable and the memory size.

Integer Data Types

Integer Data Types represent integer numbers, i.e numbers without any fractional parts or decimal points.

DataType	MemorySize	Minimum and Maximum values	DefaultValue
byte	1byte	-128to+128	0
short	2bytes	-32768to+32767	0
int	4bytes	-2147483648to+2147483647	0
long	8bytes	-92233720368547758080to+9223372036854775807	0L

Float Data Types

Float data types are represent numbers with decimal point.

DataType	MemorySize	MinimumandMaximum values	DefaultValue
float	4byte	-3.4e38to-1.4e-45and1.4e-45to3.4e38	0.0f
double	8bytes	-1.8e308to-4.9e-324and4.9e-324to1.8e308	0.0d

Note: Float data type can represent up to 7 digits accurately after decimal point.

Double data type can represent up to 15 digits accurately after decimal point.

Character Data Type

Character data type are represents a single character like a, P, &, *,..etc.

DataType	MemorySize	Minimum and Maximum values	DefaultValue
char	2bytes	0to65538	\u0000

Boolean Data Types

Boolean data types represent any of the two values, true or false. JVM uses 1 bit to represent a Boolean value internally.

DataType	MemorySize	Minimum and Maximum values	DefaultValue
boolean	1byte	0or1	0(false)

OPERATORS

An operator is a symbol that performs an operation. An operator acts on some variables called operands to get the desired result.

Example: a + b

Here a, b are operands and + is operator. Types of Operators

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment or Decrement operators
6. Conditional operator
7. Bit wise operators

1. Arithmetic Operators: Arithmetic Operators are used for mathematical calculations.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modular

Program: Java Program to implement Arithmetic Operators class ArithmeticOperators

```
{
public static void main(String[] args)
{
int a = 12, b = 5;
System.out.println("a + b = " + (a + b)); System.out.println("a - b = " + (a - b)); System.out.println("a * b = " + (a * b)); System.out.println("a / b = " + (a / b)); System.out.println("a % b = " + (a % b));
}
}
```

2. Relational Operators: Relational operators are used to compare two values and return a true or false result based upon that comparison. Relational operators are of 6 types

Operator	Description
>	Greaterthan
>=	Greaterthanorequalto
<	Lessthan
<=	Lessthanorequalto
==	Equalto
!=	Notequalto

Program: Java Program to implement Relational Operators

```
class RelationalOperator
{
public static void main(String[] args)
{
int a = 10; int b = 3; int c = 5;
System.out.println("a > b: " + (a > b));
System.out.println("a < b: " + (a < b)); System.out.println("a >= b: " + (a >= b)); System.out.println("a <=
b: " + (a <= b)); System.out.println("a == c: " + (a == c)); System.out.println("a != c: " + (a != c));
}
}
```

3. Logical Operator: The Logical operators are used to combine two or more conditions .Logical

operators are of three types

1. Logical AND (&&),
2. Logical OR (||)
3. Logician NOT (!)

1. Logical AND (&&) : Logical AND is denoted by double ampersand characters (&&).it is used to check the combinations of more than one conditions. if any one condition false the complete condition becomes false.

Truth table of Logical AND

Condition1	Condition2	Condition1&&Condition2
True	True	True
True	False	False
False	True	False
False	False	False

2. Logical OR (||) : Logical OR is denoted by double pipe characters (||). it is used to check the combinations of more than one conditions. if any one condition true the complete condition becomes true.

Truth table of Logical OR

Condition1	Condition2	Condition1&&Condition2
True	True	True
True	False	True
False	True	True
False	False	False

3. Logician NOT (!): Logical NOT is denoted by exclamatory characters (!), it is used to check the opposite result of any given test condition. i.e, it makes a true condition false and false condition true.

Truth table of Logical NOT

Condition1	!Condition2
True	False
False	True

Example of Logical Operators class LogicalOp

{

public static void main(String[] args)

{

int x=10;

System.out.println(x==10 && x>=5)); System.out.println(x==10 || x>=5)); System.out.println (! (x==10

));

}

}

4. Assignment Operator: Assignment operators are used to assign a value (or) an expression (or) a value of a variable to another variable.

Syntax : variable name=expression (or) value Example : x=10;

y=20;

The following list of Assignment operators are.

Operator	Description	Example	Meaning
+=	AdditionAssignment	x+=y	x=x+y
-=	AdditionAssignment	x-=y	x=x-y
*=	AdditionAssignment	x*=y	x=x*y
/=	AdditionAssignment	x/=y	x=x/y
%=	AdditionAssignment	x%=y	x=x%y

Example of Assignment Operators class AssignmentOperator

{

public static void main(String[] args)

{

int a = 4;

```
int var; var = a;
System.out.println("var using =: " + var); var += a;
System.out.println("var using +=: " + var); var *= a;
System.out.println("var using *=: " + var);
}
}
```

5: Increment And Decrement Operators : The increment and decrement operators are very useful. ++ and == are called increment and decrement operators used to add or subtract. Both are unary operators. The syntax of the operators is given below.

These operators in two forms : prefix (++x) and postfix(x++).

++<variable name> --<variable name>

<variable name>++ <variable name>--

Operator	Meaning
++x	PreIncrement
--x	PreDecrement
x++	PostIncrement
x--	PostDecrement

Where

1 : ++x : Pre increment, first increment and then do the operation.

2 : - -x : Pre decrement, first decrements and then do the operation.

3 : x++ : Post increment, first do the operation and then increment.

4 : x- - : Post decrement, first do the operation and then decrement.

Example

```
class Increment
{
public static void main(String[] args)
{
int var=5; System.out.println (var++); System.out.println (++var); System.out.println (var--);
System.out.println (--var);
}
}
```

6 : Conditional Operator: A conditional operator checks the condition and executes the statement depending on the condition. Conditional operator consists of two symbols.

1 : question mark (?).

2 : colon (:).

Syntax: condition ? exp1 : exp2;

It first evaluate the condition, if it is true (non-zero) then the “exp1” is evaluated, if the condition is false (zero) then the “exp2” is evaluated.

Example :

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

```
int februaryDays = 29; String result;

result = (februaryDays == 28) ? "Not a leap year" : "Leap year"; System.out.println(result);

}

}
```

7. Bitwise Operators:

- Bitwise operators are used for manipulating a data at the bit level, also called as bit level
- programming. Bit-level programming mainly consists of 0 and 1.
- They are used in numerical Computations to make the calculation process faster.
- The bitwise logical operators work on the data bit by bit.
- Starting from the least significant bit, i.e. LSB bit which is the rightmost bit, working towards the MSB (Most Significant Bit) which is the leftmost bit.

A list of Bitwise operators as follows…

Operator	Meaning
&	BitwiseAND
|	BitwiseOR
^	BitwiseXOR
~	BitwiseComplement
<<	LeftShift
>>	RightShift

1. Bitwise AND (&):

- Bitwise AND operator is represented by a single ampersand sign (&).
- Two integer expressions are written on each side of the (&) operator.
    - if any one condition false (0) the complete condition becomes false (0). Truth table of

Bitwise AND

Condition1	Condition2	Condition1&Condition2
0	0	0
0	1	0

1	0	0
1	1	1

Example :

int x = 10; int y = 20; x & y = ?

x = 0000 1010

y = 0000 1011

x & y = 0000 1010 = 10

2. Bitwise OR:

- Bitwise OR operator is represented by a single vertical bar sign (|).
- Two integer expressions are written on each side of the (|) operator.
    - if any one condition true (1) the complete condition becomes true (1). Truth table of

Bitwise OR

Condition1	Condition2	Condition1|Condition2
0	0	0
0	1	1
1	0	1
1	1	1

Example : int x = 10; int y = 20; x | y = ?

x = 0000 1010

y = 0000 1011

x | y = 0000 1011 = 11

3. Bitwise Exclusive OR :

- The XOR operator is denoted by a carrot (^) symbol.
- It takes two values and returns true if they are different; otherwise returns false.
    - In binary, the true is represented by 1 and false is represented by 0. Truth table of Bitwise

XOR

Condition1	Condition2	Condition1^Condition2
0	0	0
0	1	1
1	0	1
1	1	0

Example :

int x = 10; int y = 20; x ^ y = ?

x = 0000 1010

y = 0000 1011

x ^ y = 0000 0001 = 1

4. Bitwise Complement (~):

- The bitwise complement operator is a unary operator.
- It is denoted by ~, which is pronounced as tilde.
- It changes binary digits 1 to 0 and 0 to 1.
- bitwise complement of any integer N is equal to - (N + 1).
- Consider an integer 35. As per the rule, the bitwise complement of 35 should be -(35 + 1) = -36.

Example :

int x = 10; find the ~x value.

x = 0000 1010

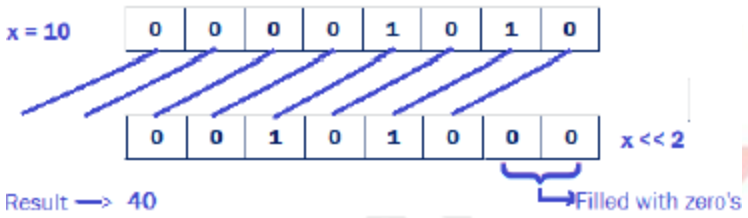
~x= 1111 0101

5. Bitwise Left Shift Operator (<<) :

- This Bitwise Left shift operator (<<) is a binary operator.
  - It shifts the bits of a number towards left a specified no.of times. Example:

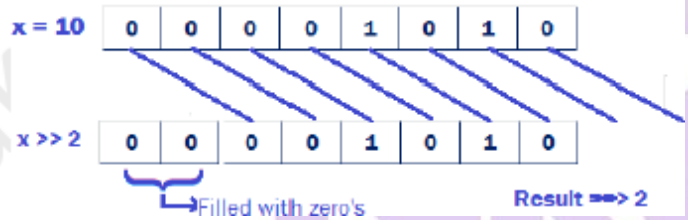
int x = 10; x << 2 = ?

6. Bitwise Right Shift Operator (>>) :



- This Bitwise Right shift operator (>>) is a binary operator.
  - It shifts the bits of a number towards right a specified no.of times. Example:

int x = 10; x >> 2 = ?



EXPRESSIONS

- In any programming language, if we want to perform any calculation or to frame any condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression. In the java programming language, an expression is defined as follows..

- An expression is a collection of operators and operands that represents a specific value.
- In the above definition, an operator is a symbol that performs tasks like arithmetic operations, logical operations, and conditional operations, etc.

Expression Types

In the java programming language, expressions are divided into THREE types. They are as follows.

- Infix Expression
- Postfix Expression
- Prefix Expression

The above classification is based on the operator position in the expression.

Infix Expression

The expression in which the operator is used between operands is called infix expression. The infix expression has the following general structure.

Example a+b

Postfix Expression

The expression in which the operator is used after operands is called postfix expression. The postfix expression has the following general structure.

Example ab+

Prefix Expression

The expression in which the operator is used before operands is called a prefix expression. The prefix expression has the following general structure.

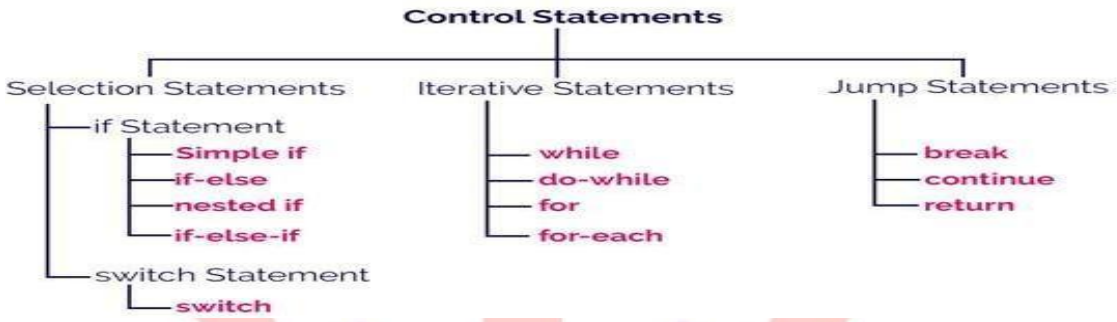
Example

+ab

CONTROL STATEMENTS

- In java, the default execution flow of a program is a sequential order.
- But the sequential order of execution flow may not be suitable for all situations.
- Sometimes, we may want to jump from line to another line, we may want to skip a part of the program, or sometimes we may want to execute a part of the program again and again.
- To solve this problem, java provides control statements.

Types of Control Statements



1. Selection Control Statements

In java, the selection statements are also known as decision making statements or branching statements. The selection statements are used to select a part of the program to be executed based on a condition.

Java provides the following selection statements.

- if statement
- if-else statement
- if-elif statement
- nested if statement
- switch statement

if statement in java

- In java, we use the if statement to test a condition and decide the execution of a block of statements based on that condition result.
- The if statement checks, the given condition then decides the execution of a block of statements. If the condition is True, then the block of statements is executed and if it is False, then the block of statements is ignored.

Syntax

```
if(condtion)
{
if-block of statements;
}
statement after if-block;
```

Example

```
public class IfStatementTest
{
public static void main(String[] args)
{
int x=10; if(x>0) x++;
System.out.println("x value is:"+x);
}
}
```

In the above execution, the number 12 is not divisible by 5. So, the condition becomes False and the condition is evaluated to False. Then the if statement ignores the execution of its block of statements.

if-else statement in java

- In java, we use the if-else statement to test a condition and pick the execution of a block of statements out of two blocks based on that condition result.
- The if-else statement checks the given condition then decides which block of statements to be executed based on the condition result.
- If the condition is True, then the true block of statements is executed and if it is False, then the false block of statements is executed.

Syntax

```
if(condtion)
{
true-block of statements;
}
else
{
false-block of statements;
}
statement after if-block;
```

Example

```
public class IfElseStatementTest
{
public static void main(String[] args)
{
int a=29;  if(a % 2==0)
System.out.println("Even Number is :"+a); else
System.out.println("Odd Number is :"+a);
}
}
```

Nested if statement in java

Writing an if statement inside another if-statement is called nested if statement.

Syntax

```
if(condition_1)
{
if(condition_2)
{
inner if-block of statements;
...
}
...
}
```

Example

```
public class NestedIfStatementTest
{
public static void main(String[] args)
{
 int num=1; if(num<10)
{
if(num==1)
{
System.out.print("The value is equal to 1);
}
else
{
System.out.print("The value is greater than 1");
```

```
}
}
else
{
System.out.print("The value is greater than 10");
}
System.out.print("Nested if - else statement ");
}
}
```

if-else if statement in java

Writing an if-statement inside else of an if statement is called if-else-if statement.

Syntax

```
if(condition_1)
{
condition_1 true-block;
...
}
else if(condition_2)
{
condition_2 true-block; condition_1 false-block too;
...
}
```

Example

```
public class IfElseIfStatementTest
{
public static void main(String[] args)
{
 int x = 30; if( x == 10 )
{
System.out.print("Value of X is 10");
}
else if( x == 20 )
{
System.out.print("Value of X is 20");
}
else if( x == 30 )
{
System.out.print("Value of X is 30");
}
else
{
System.out.print("This is else statement");
}
}
}
```


Switch

- Using the switch statement, one can select only one option from more number of options very easily.
- In the switch statement, we provide a value that is to be compared with a value associated with each option. Whenever the given value matches the value associated with an option, the execution starts from that option.
- In the switch statement, every option is defined as a case.

Syntax:

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

Example

```
class SampleSwitch
{
public static void main(String args[])
{
char color ='g'; switch(color )
{
case 'r':
System.out.println("RED") ; break ;
```

```
case 'g':
System.out.println("GREEN") ; break ; case 'b':
System.out.println("BLUE") ; break ; case 'w':
System.out.println("WHITE") ; break ; default:
System.out.println("No color") ;
}
}
}
```

2. Iteration Statements

- The java programming language provides a set of iterative statements that are used to execute a statement or a block of statements repeatedly as long as the given condition is true.
- The iterative statements are also known as looping statements or repetitive statements. Java provides the following iterative statements.

1. while statement
2. do-while statement
3. for statement
4. for-each statement

while statement in java

The while statement is used to execute a single statement or block of statements repeatedly as long as the given condition is TRUE. The while statement is also known as Entry control looping statement.

Syntax

```
while(condition)
```

```
{
// body of loop
}
```

Example

```
public class WhileTest
{
public static void main(String[] args)
{
int num = 1; while(num <= 10)
{
System.out.println(num); num++;
}
System.out.println("Statement after while!");
}
}
```

do-while statement in java

- The do-while statement is used to execute a single statement or block of statements repeatedly as long as given the condition is TRUE.
- The do-while statement is also known as the Exit control looping statement.

Syntax

```
do
{
// body of loop
```

```
} while (condition);
```

Example

```
public class DoWhileTest

{

public static void main(String[] args)

{

int num = 1; do

{

System.out.println(num); num++;

}while(num <= 10); System.out.println("Statement after do-while!");

}

}
```

for statement in java

The for statement is used to execute a single statement or a block of statements repeatedly as long as the given condition is TRUE.

Syntax

```
for(initialization; condition; inc/dec)

{

// body

}
```

If only one statement is being repeated, there is no need for the curly braces.

In for-statement, the execution begins with the initialization statement. After the initialization statement, it executes Condition. If the condition is evaluated to true, then the block of statements executed otherwise it terminates the for-statement. After the block of statements execution, the modification statement gets executed, followed by condition again.

Example

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

3. Jump Statements

The java programming language supports jump statements that used to transfer execution control from one line to another line.

The java programming language provides the following jump statements.

1. break statement
2. continue statement

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 +" ");
}
}
}
```

Elements of Java

CLASS

- In Java, classes and objects are basic concepts of Object Oriented Programming (OOPs) that are used to represent real-world concepts and entities.
- classes usually consist of two things: instance variables and methods.
- The class represents a group of objects having similar properties and behavior.
- For example, the animal type Dog is a class while a particular dog named Tommy is an object of the Dog class.
- It is a user-defined blueprint or prototype from which objects are created.
- For example, Student is a class while a particular student named Ravi is an object.
- The java class is a template of an object.
- Every class in java forms a new data type.
- Once a class got created, we can generate as many objects as we want.

Class Characteristics

Identity - It is the name given to the class.

State - Represents data values that are associated with an object. Behavior - Represents actions can be performed by an object.

Properties of Java Classes

1. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
2. Class does not occupy memory.
3. Class is a group of variables of different data types and a group of methods.
4. A Class in Java can contain:

- Data member
- Method
- Constructor
- Nested Class
- Interface

Creating a Class

In java, we use the keyword class to create a class. A class in java contains properties as variables and behaviors as methods.

Syntax

class className

{

data members declaration; methods definition;

}

- The ClassName must begin with an alphabet, and the Upper-case letter is preferred.
- The ClassName must follow all naming rules.

Example

Here is a class called Box that defines three instance variables: width, height, and depth. class Box

{

double width; double height; double depth; void volume()

{

…….

}

}

OBJECT

- In java, an object is an instance of a class.
- Objects are the instances of a class that are created to use the attributes and methods of a class.
- All the objects that are created using a single class have the same properties and methods. But the value of properties is different for every object.

Syntax

ClassName objectName = new ClassName( );

- The objectName must begin with an alphabet, and a Lower-case letter is preferred.
- The objectName must follow all naming rules.

Example

Box mybox = new Box();

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

- A method is a block of statements under a name that gets executes only when it is called.
- Every method is used to perform a specific task. The major advantage of methods is code reusability (define the code once, and use it many times).
- In a java programming language, a method defined as a behavior of an object. That means, every method in java must belong to a class.
- Every method in java must be declared inside a class.
- Every method declaration has the following characteristics.
- returnType - Specifies the data type of a return value.
- name - Specifies a unique name to identify it.
- parameters - The data values it may accept or recieve.
- { } - Defienes the block belongs to the method.

Creating a method

A method is created inside the class

Syntax

```
class ClassName

{

returnType methodName( parameters )

{

// body of method

}

}
```

Calling a method

- In java, a method call precedes with the object name of the class to which it belongs and a dot operator.
- It may call directly if the method defined with the static modifier.
- Every method call must be made, as to the method name with parentheses (), and it must

terminate with a semicolon.

Syntax

objectName.methodName(actualArguments);

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();

}

}
```

CONSTRUCTORS

- Constructor in Java is a special member method which will be called automatically by the JVM whenever an object is created for placing user 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. Initialization is a process of assigning user defined values at the time of allocation of memory space.

Syntax

```
ClassName()

{

.....

.....

}
```

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 then 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 variables 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 TestDemo
{
public static void main(String args[])
{
Test t1=new Test(10, 20);
}
}
```

ACCESS CONTROL(MEMBER ACCESS)

In Java, Access modifiers help to restrict the scope of a class, constructor, variable, method, or data member. It provides security, accessibility, etc to the user depending upon the access modifier used with the element.

Types of Access Modifiers in Java

There are four types of access modifiers available in Java:

1. Default – No keyword required
2. Private
3. Protected
4. Public

1. Default Access Modifier

- When no access modifier is specified for a class, method, or data member – It is said to be having the default access modifier by default.
- The default modifier is accessible only within package.
- It cannot be accessed from outside the package.
- It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

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

{

void msg()

{

System.out.println("Hello");

}

}

//save by B.java package mypack;
```

```
import pack.*; class B

{

public static void main(String args[])

{

A obj = new A(); //Compile Time Error obj.msg(); //Compile Time Error

}

}
```

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

2. Private

- The private access modifier is accessible only within the class.
- The private access modifier is specified using the keyword private.
- The methods or data members declared as private are accessible only within the class in which they are declared.
- Any other class of the same package will not be able to access these members.
- Top-level classes or interfaces can not be declared as private because private means “only visible within the enclosing class”.

Example

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

```
class A

{

private int data=40; private void msg()
```

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

3. Protected

- The protected access modifier is accessible within package and outside the package but through inheritance only.
  - The protected access modifier is specified using the keyword protected. Example
- In this example, we have created the two packages pack and mypack.
- The A class of pack package is public, so can be accessed from outside the package.
- But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java package pack; public class A
{
protected void msg()
```

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

4. Public

- The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.
- The public access modifier is specified using the keyword public.

Example

```
//save by A.java package pack; public class A
{
public void msg()
```

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

Table: Class Member Access

Let'sunderstandtheacce ssmodifiersinJavabyasi mpletable.Access Modifier	within class	withinpackage	outsidepackage bysubclassonly	outside package
Private	YES	NO	NO	NO
Default	YES	YES	NO	NO
Protected	YES	YES	YES	NO
Public	YES	YES	YES	YES

GENERICS IN JAVA

The java generics is a language feature that allows creating methods and class which can handle any type of data values. The generic programming is a way to write generalized programs, java supports it by java generics.

The java generics is similar to the templates in the C++ programming language.

- Most of the collection framework classes are generic classes.
- The java generics allows only non-primitive type, it does not support primitive types like int, float, char, etc.

The java generics feature was introduced in Java 1.5 version. In java, generics used angular brackets "< >". In java, the generics feature implemented using the following.

- **Generic Method**
- **Generic Classe**

Generic methods in Java

The java generics allows creating generic methods which can work with a different type of data values. Using a generic method, we can create a single method that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.

Let's look at the following example program for generic method.

Example - Generic method

```
public class GenericFunctions {
	public <T, U> void displayData(T value1, U value2) {
		System.out.println("(" + value1.getClass().getName() + ", " + value2.getClass().getName() +
")");
	}

	public static void main(String[] args) {
		GenericFunctions obj = new GenericFunctions();

		obj.displayData(45.6f, 10);
		obj.displayData(10, 10);
		obj.displayData("Hi", 'c');
	}

}
```

In the above example code, the method displayData( ) is a generic method that allows a different type of parameter values for every function call.

Generic Class in Java

In java, a class can be defined as a generic class that allows creating a class that can work with different types.

A generic class declaration looks like a non-generic class declaration, except that the class name is

followed by a type parameter section.
Let's look at the following example program for generic class.

Example - Generic class

```
public class GenericsExample<T> {
	T obj;
	public GenericsExample(T anotherObj) {
		this.obj = anotherObj;
	}
	public T getData() {
		return this.obj;
	}

	public static void main(String[] args) {

		GenericsExample<Integer> actualObj1 = new GenericsExample<Integer>(100);
		System.out.println(actualObj1.getData());

		GenericsExample<String> actualObj2 = new GenericsExample<String>("Java");
		System.out.println(actualObj2.getData());

		GenericsExample<Float> actualObj3 = new GenericsExample<Float>(25.9f);
		System.out.println(actualObj3.getData());
	}
}
```

INNER CLASSES

- Inner class means one class which is a member of another class.
- We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Syntax of Inner class

```
class Outer_class
{
//code
class Inner_class
{
//code
}
}
```

Types of Inner classes

There are four types of inner classes.

1. Member Inner class
2. Local inner classes
3. Anonymous inner classes
4. Static nested classes

1. MEMBER INNER CLASS

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

```
class Outer
{
//code
```

```
class Inner
{
//code
}
}
```

Example

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

2. ANONYMOUS INNER CLASS

- In Java, a class can contain another class known as nested class. It's possible to create a nested class without giving any name.
- A nested class that doesn't have any name is known as an anonymous class.
- An anonymous class must be defined inside another class. Hence, it is also known as an anonymous inner class.

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();
}
}
```

1. A class is created, but its name is decided by the compiler, which extends the Person class and provides the implementation of the eat() method.

2. An object of the Anonymous class is created that is referred to by 'p,' a reference variable of Person type.

3. 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();
```

```
}

}
```

4. STATIC NESTED CLASS

- A static class i.e. created inside a class is called static nested class in java. It cannot access nonstatic 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();

}

}
```

In this example, you need to create the instance of static nested class because it has instance method msg(). But you don't need to create the object of Outer class because nested class is static and static properties, methods or classes can be accessed without object.

STRING CLASS

- A string is a sequence of characters surrounded by double quotations. In a java programming language, a string is the object of a built-in class String.
- The string created using the String class can be extended. It allows us to add more characters after its definition, and also it can be modified.

Example

String siteName = "javaprogramming"; siteName = "javaprogramminglanguage"; **String handling**

methods

In java programming language, the String class contains various methods that can be used to handle string data values.

The following table depicts all built-in methods of String class in java.

S. No	Method	Description
1	charAt(int)	Findsthecharacteratgivenindex
2	length()	Findsthelengthofgivenstring
3	compareTo(String)	Comparestwostrings
4	compareToIgnoreCase(String)	Comparestwostrings,ignoringcase
5	concat(String)	Concatenatestheobjectstringwithargumentstring.
6	contains(String)	Checkswhetherastringcontainssub-string
7	contentEquals(String)	Checkswhethertwostringsaresame
8	equals(String)	Checkswhethertwostringsaresame
9	equalsIgnoreCase(String)	Checkswhethertwostringsaresame,ignoringcase
10	startsWith(String)	Checkswhetherastringstartswiththespecifiedstring
11	isEmpty()	Checkswhetherastringisemptyornot
12	replace(String,String)	Replacesthefirststringwithsecondstring

13	replaceAll(String,String)	Replacesthefirststringwithsecondstringatall occurrences.
14	substring(int,int)	Extractsasub-stringfromspecifiedstartandendindex values
15	toLowerCase()	Convertsastringtolowercaseletters
16	toUpperCase()	Convertsastringtouppercaseletters
17	trim()	Removeswhitespacefrombothends
18	toString(int)	ConvertsthevaluetoaStringobject

Example

```
public class JavaStringExample
{
public static void main(String[] args)
{
String title = "Java Programming";
String siteName = "String Handling Methods"; System.out.println("Length of title: " + title.length());
System.out.println("Char at index 3: " + title.charAt(3)); System.out.println("Index of 'T': " +
title.indexOf('T')); System.out.println("Empty: " + title.isEmpty()); System.out.println("Equals: " +
siteName.equals(title)); System.out.println("Sub-string: " + siteName.substring(9, 14));
System.out.println("Upper case: " + siteName.toUpperCase());
}
}
```

JAVA ANNOTATIONS

- Java Annotation is a tag that represents the metadata i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.
- Annotations in Java are used to provide additional information, so it is an alternative option for XML and Java marker interfaces.

Example

```
@Override

@SuppressWarnings @Deprecated
```

@Override

- @Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.
- Sometimes, we does the silly mistake such as spelling mistakes etc. So, it is better to mark @Override annotation that provides assurity that method is overridden.

Example

```
class Animal

{
void eatSomething()

{

System.out.println("eating something");}

}

class Dog extends Animal

{

@Override

void eatsomething()

{

System.out.println("eating foods");

} //should be eatSomething

}

class TestAnnotation1
```

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

Output: Comple Time Error

@Deprecated

@Deprecated annotation marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions. So, it is better not to use such methods.

Example

Output

```
class A
{
void m()
{
System.out.println("hello m");
}
@Deprecated void n()
{
System.out.println("hello n");
}
}
class TestAnnotation3
{
public static void main(String args[])
{
 A a=new A(); a.n();
}
}
```

At Compile Time:

Note: Test.java uses or overrides a deprecated API. Note: Recompile with -Xlint:deprecation for details.

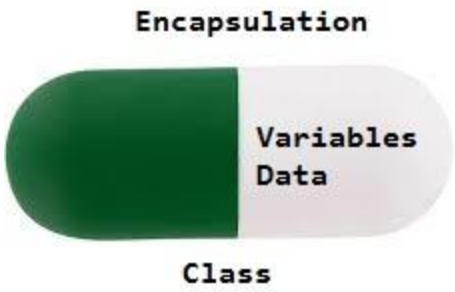
At Runtime:

hello n

OOP Principles:

Encapsulation

- Encapsulation is the process of combining data and code into a single unit.
- In OOP, every object is associated with its data and code.
- In programming, data is defined as variables and code is defined as methods.
- The java programming language uses the class concept to implement encapsulation.
- **Explanation:** In this example, the class restricts direct access to it from outside. The `getName()` and `setName()` methods, known as getters and setters, provide controlled access to the `name` attribute. This encapsulation mechanism protects the internal state of the `Programmer` object and allows for better control and flexibility in how the `name` attribute is accessed and modified.



'this' KEYWORD IN JAVA

this is a reference variable that refers to the current object. It is a keyword in java language represents

current class object

Why use this keyword in java ?

- The main purpose of using this keyword is to differentiate the formal parameter and data members of class, whenever the formal parameter and data members of the class are similar then JVM get ambiguity (no clarity between formal parameter and member of the class).
- To differentiate between formal parameter and data member of the class, the data member of the class must be preceded by "this".

Syntax: this.data member of current class. **Example without using this keyword** class Employee

{

```
int id;
String name;
Employee(int id,String name)
{
id = id;
name = name;
}
void show()
{
System.out.println(id+" "+name);
}
}
class ThisDemo1
```

```
{
public static void main(String args[])
{
Employee e1 = new Employee(111,"Harry"); e1.show();
}
}
```

Output: 0 null

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

Example of this keyword in java

```
class Employee
{
int id;
String name;
Employee(int id,String name)
{
this.id = id; this.name = name;
}
void show()
{
System.out.println(id+" "+name);
}
class ThisDemo2
```

```
{
public static void main(String args[])
{
Employee e1 = new Employee(111,"Harry"); e1.show();
}
}
```

Output: 111 Harry

INHERITANCE IN JAVA

- Inheritance is an important pillar of OOP(Object-Oriented Programming).
- The process of obtaining the data members and methods from one class to another class is known as inheritance.

Important Terminologies Used in Java Inheritance

Super Class/Parent Class: The class whose features are inherited is known as a superclass(or a base class or a parent class).

Sub Class/Child Class: The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

Why Do We Need Java Inheritance?

Code Reusability: The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.

Method Overriding: Method Overriding is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.

Abstraction: The concept of abstract where we do not have to provide all details is achieved through inheritance. Abstraction only shows the functionality to the user.

How to Use Inheritance in Java?

- The extends keyword is used for inheritance in Java.

- Using the extends keyword indicates you are deriving from an existing class. In other words, “extends” refers to increased functionality.

Syntax

class SubclassName extends SuperclassName

{

//methods and fields

}

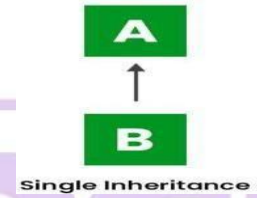
TYPES OF INHERITANCE

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. Multilevel inheritance
3. Hierarchical inheritance
4. Multiple inheritance
5. Hybrid inheritance

1. Single inheritance

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



Example

class Animal

{

String name; void show()

{

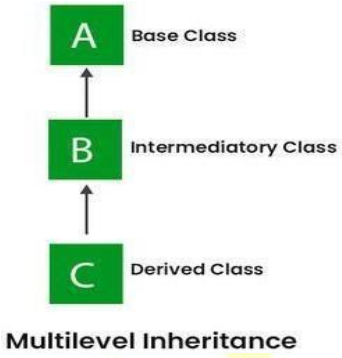
System.out.println("Animal name is:"+name);

}

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

2. Multilevel inheritances in Java

- When there is a chain of inheritance, it is known as multilevel inheritance.
- In Multilevel inheritances there exists single base class, single derived class and multiple intermediate base classes.



Multilevel Inheritance

Example

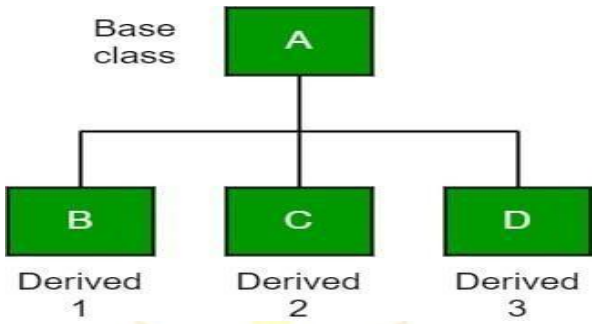
In the example, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

```
class Animal
{
String name; void show()
{
System.out.println("Animal Name is"+name);
}
}
class Dog extends Animal
{
void bark()
{
System.out.println("Mother Dog Barking...");
}
}
class BabyDog extends Dog
```

```
{
void weep()
{
System.out.println("Baby Dog weeping");
}
}
class TestInheritance2
{
public static void main(String args[])
{
BabyDog d=new BabyDog(); d.name="MotherDog"; d.show();
d.bark();
d.weep();
}
}
```

3. Hierarchical Inheritance

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived class B, C and D.



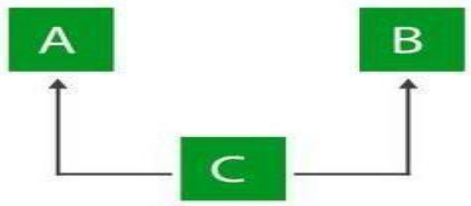
Example

```
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();
//c.bark();//C.T.Error
}
}
```

4. Multiple inheritance

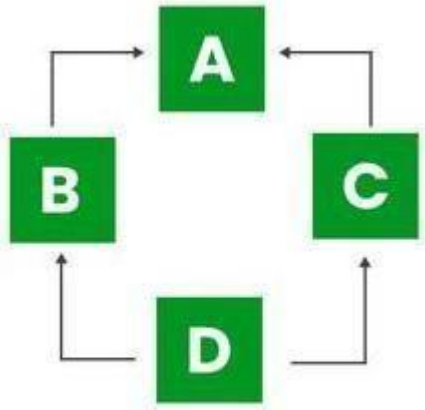
In multiple inheritance there exist multiple classes and single derived class.



The concept of multiple inheritance is not supported in java through concept of classes but it can be supported through the concept of interface.

5. Hybrid inheritance

It is a mix of two or more of the above types of inheritance. Since Java doesn't support multiple inheritances with classes, hybrid inheritance is also not possible with classes. In Java, we can achieve hybrid inheritance only through Interfaces.



Hybrid Inheritance

SUBSTITUTABILITY

- The inheritance concept used for the number of purposes in the java programming language. One of the main purposes is substitutability.
- The substitutability means that when a child class acquires properties from its parent class, the object of the parent class may be substituted with the child class object.
- For example, if B is a child class of A, anywhere we expect an instance of A we can use an instance of B.
- The substitutability can achieve using inheritance, whether using extends or implements keywords.

FORMS OF INHERITANCE

The following are the different forms of inheritance in java.

- Specialization
- Specification
- Construction
- Extension
- Limitation
- Combination

Specialization

It is the most ideal form of inheritance. The subclass is a special case of the parent class.
It holds the principle of substitutability.

Specification

This is another commonly used form of inheritance. In this form of inheritance, the parent class just specifies which methods should be available to the child class but doesn't implement them. The java provides concepts like abstract and interfaces to support this form of inheritance. It holds the principle of substitutability.

Construction

This is another form of inheritance where the child class may change the behavior defined by the parent class (overriding). It does not hold the principle of substitutability.

Extension

This is another form of inheritance where the child class may add its new properties. It holds the principle of substitutability.

Limitation

This is another form of inheritance where the subclass restricts the inherited behavior. It does not hold the principle of substitutability.

Combination

This is another form of inheritance where the subclass inherits properties from multiple parent classes. Java does not support multiple inheritance type.

BENEFITS OF INHERITANCE

- Inheritance helps in code reuse. The child class may use the code defined in the parent class without re-writing it.
- Inheritance can save time and effort as the main code need not be written again.
- Inheritance provides a clear model structure which is easy to understand.
- An inheritance leads to less development and maintenance costs.
- With inheritance, we will be able to override the methods of the base class so that the meaningful implementation of the base class method can be designed in the derived class. An inheritance leads to less development and maintenance costs.
- In inheritance base class can decide to keep some data private so that it cannot be altered by the derived class.

THE COSTS OF INHERITANCE

- Inheritance decreases the execution speed due to the increased time and effort it takes, the program to jump through all the levels of overloaded classes.
- Inheritance makes the two classes (base and inherited class) get tightly coupled. This means one cannot be used independently of each other.

- The changes made in the parent class will affect the behavior of child class too.
- The overuse of inheritance makes the program more complex.

ACCESS CONTROL(MEMBER ACCESS)

In Java, Access modifiers help to restrict the scope of a class, constructor, variable, method, or data member. It provides security, accessibility, etc to the user depending upon the access modifier used with the element.

Types of Access Modifiers in Java

There are four types of access modifiers available in Java:

1. Default – No keyword required
2. Private
3. Protected
4. Public

1. Default Access Modifier

- When no access modifier is specified for a class, method, or data member – It is said to be having the default access modifier by default.
- The default modifier is accessible only within package.
- It cannot be accessed from outside the package.
- It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java package pack; class A
{
void msg()
```

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

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

2. private

- The private access modifier is accessible only within the class.
- The private access modifier is specified using the keyword private.
- The methods or data members declared as private are accessible only within the class in which they are declared.
- Any other class of the same package will not be able to access these members.
- Top-level classes or interfaces can not be declared as private because private means “only visible within the enclosing class”.

Example

- In this example, we have created two classes A and Simple.

- A class contains private data member and private method.
- We are accessing these private members from outside the class, so there is a compile-time error.

```
class A
{
private int data=40; private void msg()
{
System.out.println("Hello java");}
}

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

3. protected

- The protected access modifier is accessible within package and outside the package but through inheritance only.
  - The protected access modifier is specified using the keyword protected. Example
- In this example, we have created the two packages pack and mypack.

- The A class of pack package is public, so can be accessed from outside the package.
- But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

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

4. public

- The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.

- The public access modifier is specified using the keyword public.

Example

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

Table: class member access

Let's understand the access modifiers in Java by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	YES	NO	NO	NO
Default	YES	YES	NO	NO
Protected	YES	YES	YES	NO
Public	YES	YES	YES	YES

SUPER KEYWORD

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

Usage of Java super Keyword

1. Super keyword At Variable Level
2. Super keyword At Method Level
3. Super keyword At Constructor Level

- Whenever the derived class is inherits the base class features, there is a possibility that base class features are similar to derived class features and JVM gets an ambiguity.
- In order to differentiate between base class features and derived class features must be preceded by super keyword.

Syntax

super.baseclass features

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 Animal
{
String color="white";
}
class Dog extends Animal
{
String color="black"; void printColor()
{
System.out.println(color); //prints color of Dog class System.out.println(super.color); //prints color of
Animal class
}
}
class TestSuper1
{
public static void main(String args[])
{
Dog d=new Dog(); d.printColor();
}
}
```

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 Animal
{
void eat()
{
System.out.println("eating...");
}
}
class Dog extends Animal
{
void eat()
{
System.out.println("eating bread...");
}
void dispay()
{
eat();
super.eat();
}
}
class TestSuper2
```

```
{
public static void main(String args[])
{
Dog d=new Dog(); d.display();
}
}
```

3. Super keyword At Constructor Level

The super keyword can also be used to invoke the parent class constructor. class Animal

```
{
Animal()
{
System.out.println("animal is created");
}
}
class Dog extends Animal
{
Dog()
{
super();
System.out.println("dog is created");
}
}
```

```
class TestSuper3
{
public static void main(String args[])
{
Dog d=new Dog();
}
}
```

FINAL KEYWORD

- It is used to make a variable as a constant, Restrict method overriding, Restrict inheritance.
- Final keyword is used to make a variable as a constant.
- This is similar to const in other language.

In java language final keyword can be used in following ways:

1. Final Keyword at Variable Level
2. Final Keyword at Method Level
3. Final Keyword at Class Level

1. Final at variable level

- A variable declared with the final keyword cannot be modified by the program after initialization.
- This is useful to universal constants, such as "PI".

Example

```
class Bike
{
final int speedlimit=90; void run()
{
speedlimit=400;
```

```
}
public static void main(String args[])
{
Bike9 obj=new Bike9(); obj.run();
}
}
```

Output: Compile Time Error

2. Final Keyword at method level

- It makes a method final, meaning that sub classes can not override this method. The compiler checks and gives an error if you try to override the method.
- When we want to restrict overriding, then make a method as a final.

Example

```
class Bike
                                                                                {
final void run()
                                                                                {
System.out.println("running");
                                                                                }
                                                                                }
class Honda extends Bike
                                                                                {
void run()
                                                                                {
System.out.println("running safely with 100kmph");
```

```
}
public static void main(String args[])
{
Honda honda= new Honda(); honda.run();
}
}
```

Output: It gives an error

3. Final Keyword at Class Level

It makes a class final, meaning that the class cannot be inheriting by other classes. When we want to restrict inheritance then make class as a final.

Example

```
final class Bike
                                                                    {
                                                                    }
class Honda1 extends Bike
                                                                    {
void run()
                                                                    {
System.out.println("running safely with 100kmph");
                                                                    }
public static void main(String args[])
{
Honda1 honda= new Honda1(); honda.run();
```

```
}

}
```

Output: Compile Time Error

POLYMORPHISM

- The polymorphism is the process of defining same method with different implementation. That means creating multiple methods with different behaviors.

Types of Java polymorphism

The Java polymorphism is mainly divided into two types:

1. Compile-time Polymorphism(Method Overloading)
2. Runtime Polymorphism(Method Overriding)

Ad Hoc Polymorphism(Method Overloading)

Whenever same method name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as method overloading.

Example

```
class Addition

{

void sum(int a, int b)

{

System.out.println(a+b);

}

void sum(int a, int b, int c)

{

System.out.println(a+b+c);

}

void sum(float a, float b)
```

```
{
System.out.println(a+b);
}
}
class Methodload
{
public static void main(String args[])
{
Addition obj=new Addition(); obj.sum(10, 20);
obj.sum(10, 20, 30);
obj.sum(10.05f, 15.20f);
}
}
```

Pure Polymorphism(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.
- In a java programming language, pure polymorphism carried out with a method overriding concept.

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"); super.walk();
}
}
class OverridingDemo
{
public static void main(String args[])
{
Man obj = new Man(); obj.walk();
}
}
```

Type Casting

- When a data type is converted into another data type by a programmer using the casting operator while writing a program code, the mechanism is known as type casting.
- In typing casting, the destination data type may be smaller than the source data type when converting the data type to another data type, that's why it is also called narrowing conversion.

Syntax

destination_datatype = (target_datatype)variable; (): is a casting operator.

target_datatype : is a data type in which we want to convert the source data type.

Example float x; byte y; y=(byte)x;

Program

public class NarrowingTypeCastingExample

```
{
public static void main(String args[])
{
double d = 166.66; int i = (int)d;
System.out.println("Before conversion: "+d); System.out.println("After conversion into int type: "+i);
}
}
```

Output

Before conversion: 166.66

After conversion into int type: 166

ABSTRACT CLASS

- A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).
- An abstract class must be declared with an abstract keyword.
- It cannot be instantiated. It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

There are two ways to achieve abstraction in java.

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

Syntax

abstract class className

{

.....

}

ABSTRACT METHOD

- An abstract method 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 class Shape

{

abstract void draw();

}

```
class Rectangle extends Shape
{
void draw()
{
System.out.println("drawing  rectangle");
}
}
class Circle1 extends Shape
{
void draw()
{
System.out.println("drawing  circle");
}
}
class TestAbstraction1
{
static void main(String args[])
{
Shape s=new Circle1(); s.draw();
}
}
```


Example2

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

```
abstract class Shape
{
int length, breadth, radius;
Scanner input = new Scanner(System.in); abstract void printArea();
}
class Rectangle extends Shape
{
void printArea()
{
System.out.println("*** Finding the Area of Rectangle ***"); System.out.print("Enter length and breadth:
");
length = input.nextInt(); breadth = input.nextInt();
System.out.println("The area of Rectangle is: " + length * breadth);
}
}
class Triangle extends Shape
{
void printArea()
{
System.out.println("\n*** Finding the Area of Triangle ***"); System.out.print("Enter Base And Height:
");
length = input.nextInt();
```

```
breadth = input.nextInt();
System.out.println("The area of Triangle is: " + (length * breadth) / 2);
}
}
class Cricle extends Shape
{
void printArea()
{
System.out.println("\n*** Finding the Area of Cricle ***"); System.out.print("Enter Radius: ");
radius = input.nextInt();
System.out.println("The area of Cricle is: " + 3.14f * radius * radius);
}
}
public class AbstractClassExample
{
public static void main(String[] args)
{
Rectangle rec = new Rectangle(); rec.printArea();
Triangle tri = new Triangle(); tri.printArea();
Cricle cri = new Cricle(); cri.printArea();
```

```
}

}
```

OBJECT CLASS

- In java, the Object class is the super most class of any class hierarchy. The Object class in the java programming language is present inside the java.lang package.
- Every class in the java programming language is a subclass of Object class by default.
- The Object class is useful when you want to refer to any object whose type you don't know. Because it is the superclass of all other classes in java, it can refer to any type of object.

Method	**Description**	**ReturnValue**
getClass()	Returns Class class object	object
hashCode()	returns the hashcode number for object being used.	int
equals(Object obj)	compares the argument object to calling object.	boolean
clone()	Compares two strings, ignoring case	int
concat(String)	Creates copy of invoking object	object
toString()	returns the string representation of invoking object.	String
notify()	wakes up a thread, waiting on invoking object's monitor.	void
notifyAll()	wakes up all the threads, waiting on invoking object's monitor.	void
wait()	causes the current thread to wait, until another thread notifies.	void
wait(long,int)	causes the current thread to wait for the specified millisecondsand nanoseconds, until another thread notifies.	void
finalize()	It is invoked by the garbage collector before an object is being garbage collected.	void

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.

Why do we use an Interface?

- It is used to achieve total abstraction.

- Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.
- Any class can extend only 1 class but can any class implement infinite number of interface.
- Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?
- The reason is, abstract classes may contain non-final variables, whereas variables in the interface are final, public and static.

DIFFERENCE BETWEEN CLASS AND INTERFACE

Class	**Interface**
The keyword used to create a class is "class"	The keyword used to create an interface is "interface"
A class can be instantiated i.e., objects of a class can be created.	An Interface cannot be instantiated i.e. objects cannot be created.
Classes do not support multiple inheritance.	The interface supports multiple inheritance.
It can be inherited from another class.	It cannot inherit a class.
It can be inherited by another class using the keyword 'extends'.	It can be inherited by a class by using the keyword 'implements' and it can be inherited by an interface using the keyword 'extends'.
It can contain constructors.	It cannot contain constructors.
It cannot contain abstract methods.	It contains abstract methods only.
Variables and methods in a class can be declared using any access specifier(public, private, default, protected).	All variables and methods in an interface aredeclared as public.
Variables in a class can be static, final, or neither.	All variables are static and final.

DEFINING INTERFACES

The interface keyword is used to declare an interface.

Syntax

```
interface interface_name
{
declare constant fields declare methods that abstract
}
```

Example

```
interface A
{
public static final int a = 10; void display();
}
```

IMPLEMENTING INTERFACES

A class uses the implements keyword to implement an interface.

Example

```
interface A
{
public static final int a = 10; void display();
}
class B implements A
{
public void display()
{
```

```
System.out.println("Hello");
}
}
class InterfaceDemo
{
public static void main (String[] args)
{
B obj= new B(); obj.display(); System.out.println(a);
}
}
```

APPLYING INTERFACES

To understand the power of interfaces, let's look at a more practical example.

Example:

```
interface IntStack
{
void push(int item); int pop();
}
class FixedStack implements IntStack
{
private int stck[]; private int top;
```

```
FixedStack(int size)
{
stck = new int[size]; top = -1;
}
public void push(int item)
{
if(top==stck.length-1) System.out.println("Stack is full."); else
stck[++top] = item;
}
public int pop()
{
if(top ==-1)
{
System.out.println("Stack underflow."); return 0;
}
else
return stck[top--];
}
}
class InterfaceTest
```

```
{
public static void main(String args[])
{
FixedStack mystack1 = new FixedStack(5); FixedStack mystack2 = new FixedStack(8); for(int i=0; i<5;
i++)
 mystack1.push(i); for(int i=0; i<8; i++) mystack2.push(i); for(int i=0; i<5; i++)
System.out.println(mystack1.pop()); for(int i=0; i<8; i++) System.out.println(mystack2.pop());
}
}
```

VARIABLES IN INTERFACE

- Variables can be declared inside of interface declarations. They are implicitly final and static, meaning they cannot be changed by the implementing class.
- You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values.

Example

```
interface SharedConstants
{
int NO = 0; int YES = 1;
int MAYBE = 2;
```

```
int LATER = 3; int NEVER = 4;
}
class Question implements SharedConstants
{
BufferedReader br=new BufferedReader(new InputStreamreader(System.in)); int ask()
{
System.out.println("would u like to have a cup of coffee?) String ans=br.readLine();
if (ans= ="no") return NO;
else if (ans=="yes") return YES;
else if (ans=="notnow") return LATER;
else
return NEVER;
}
}
class AskMe
{
public static void main(String args[])
{
```

```
Question q = new Question(); System.out.println(q.ask());

}

}
```

EXTENDING INTERFACES

- One interface can inherit another by use of the keyword extends.
- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

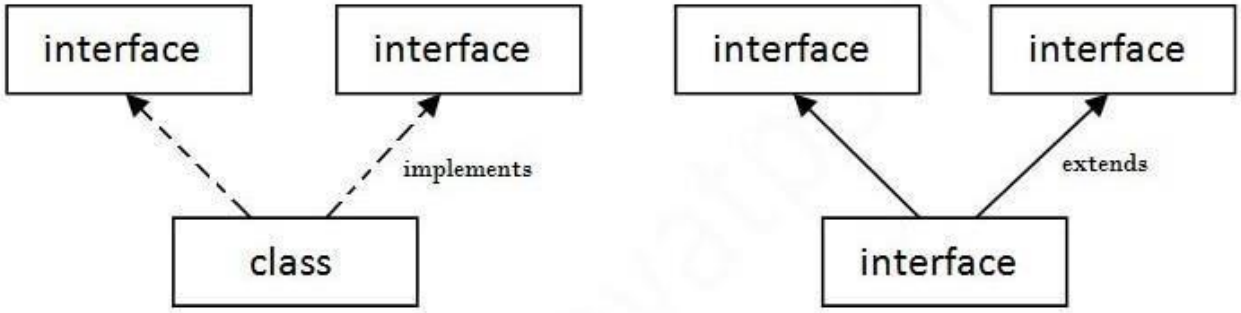
Example

```
interface A

{

void meth1(); void meth2();

}

interface B extends A

{

void meth3();

}

class MyClass implements B

{

public void meth1()

{

System.out.println("Implement  meth1().");

}

public void meth2()

{
```

```
System.out.println("Implement  meth2().");
}
public void meth3()
{
System.out.println("Implement  meth3().");
}
}
class InterfaceDemo
{
public static void main(String args[])
{
MyClass ob = new MyClass(); ob.meth1();
ob.meth2();
ob.meth3();
}
}
```

MULTIPLE INHERITANCE IN JAVA BY INTERFACE

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

Example

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

```
}
}
```

Unit -2

Exception Handling: Exception and Error, Exception Types, Exception Handler, Exception Handling Clauses – try, catch, finally, throws and the throw statement, Built-in-Exceptions and Custom Exceptions. Files and I/O Streams: The file class, Streams, The Byte Streams, Filtered Byte Streams, The Random-Access File class

EXCEPTION

- An Exception is a run time error, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- It is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions.
- Exceptions can be caught and handled by the program.

When an exception occurs within a method, it creates an object. This object is called the exception object

- It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

Major reasons why an exception Occurs

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out of disk memory)
- Code errors
- Opening an unavailable file

Errors

- Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.
- Errors are usually beyond the control of the programmer, and we should not try to handle errors.

EXCEPTION HANDLING IN JAVA

An exception in java programming is an abnormal situation that is araised during the program execution. In

simple words, an exception is a problem that arises at the time of program execution.
When an exception occurs, it disrupts the program execution flow. When an exception occurs, the program execution gets terminated, and the system generates an error. We use the exception handling mechanism to avoid abnormal termination of program execution.
Java programming language has a very powerful and efficient exception handling mechanism with a large number of built-in classes to handle most of the exceptions automatically.
Java programming language has the following class hierarchy to support the exception handling mechanism.

EXCEPTION TYPES IN JAVA

In java, exceptions are mainly categorized into two types, and they are as follows.

- **Checked Exceptions**
- **Unchecked Exceptions**

Checked Exceptions
The checked exception is an exception that is checked by the compiler during the compilation process to confirm whether the exception is handled by the programmer or not. If it is not handled, the compiler displays a compilation error using built-in classes.
The checked exceptions are generally caused by faults outside of the code itself like missing resources, networking errors, and problems with threads come to mind.
The following are a few built-in classes used to handle checked exceptions in java.

- IOException
- FileNotFoundException
- ClassNotFoundException
- SQLException
- DataAccessException
- InstantiationException
- UnknownHostException

In the exception class hierarchy, the checked exception classes are the direct children of the Exception class.
The checked exception is also known as a compile-time exception.

Unchecked Exceptions
The unchecked exception is an exception that occurs at the time of program execution. The unchecked exceptions are not caught by the compiler at the time of compilation.
The unchecked exceptions are generally caused due to bugs such as logic errors, improper use of resources, etc.
The following are a few built-in classes used to handle unchecked exceptions in java.

- ArithmeticException
- NullPointerException
- NumberFormatException
- ArrayIndexOutOfBoundsException
- StringIndexOutOfBoundsException

In the exception class hierarchy, the unchecked exception classes are the children of RuntimeException class, which is a child class of Exception class.
The unchecked exception is also known as a runtime exception.

Exception Handling Clauses:
try AND catch IN JAVA

In java, the **try** and **catch**, both are the keywords used for exception handling.

The keyword try is used to define a block of code that will be tests the occurence of an exception. The keyword catch is used to define a block of code that handles the exception occured in the respective try block.

The uncaught exceptions are the exceptions that are not caught by the compiler but automatically caught and handled by the Java built-in exception handler.

Both try and catch are used as a pair. Every try block must have one or more catch blocks. We can not use try without atleast one catch, and catch alone can be used (catch without try is not allowed).

The following is the syntax of try and catch blocks.

Syntax

```
try{
   ...
   code to be tested
   ...
}
catch(ExceptionType object){
   ...
   code for handling the exception
   ...
}
```

throw, throws, AND finally KEYWORDS IN JAVA

In java, the keywords throw, throws, and finally are used in the exception handling concept. Let's look at each of these keywords.

throw keyword in Java

The throw keyword is used to throw an exception instance explicitly from a try block to corresponding catch block. That means it is used to transfer the control from try block to corresponding catch block.

The throw keyword must be used inside the try blcok. When JVM encounters the throw keyword, it stops the execution of try block and jump to the corresponding catch block.

- Using throw keyword only object of Throwable class or its sub classes can be thrown.
- Using throw keyword only one exception can be thrown.
- The throw keyword must followed by an throwable instance.

The following is the general syntax for using throw keyword in a try block.

Syntax

```
throw instance;
```

Here the instace must be throwable instance and it can be created dynamically using new operator

throws keyword in Java

The throws keyword specifies the exceptions that a method can throw to the default handler and does not handle itself. That means when we need a method to throw an exception automatically, we use throws keyword followed by method declaration

- When a method throws an exception, we must put the calling statement of method in try-catch block.

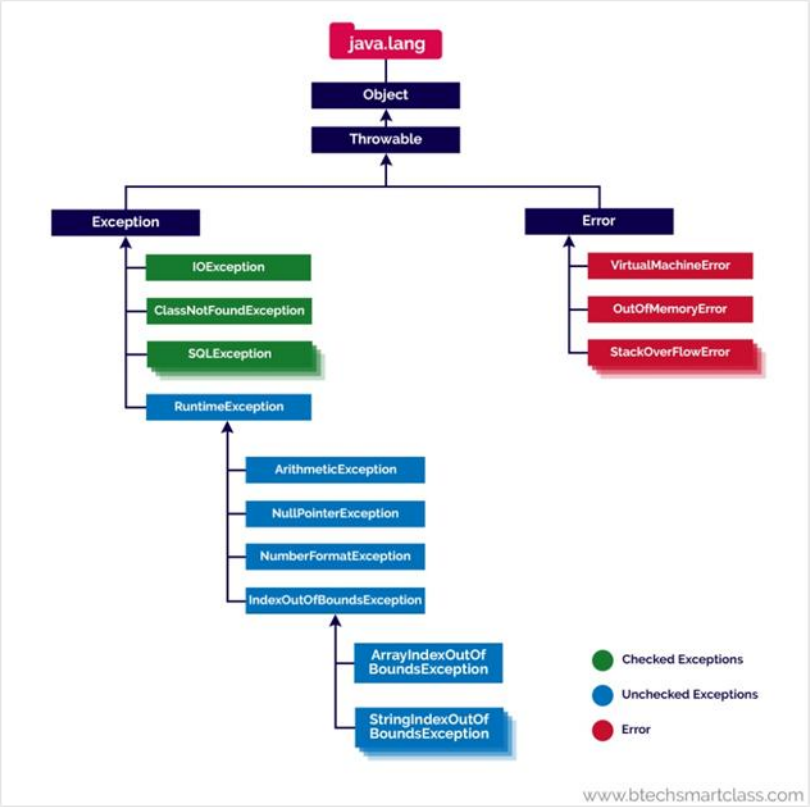
finally keyword in Java

The finally keyword used to define a block that must be executed irrespective of exception occurence.

The basic purpose of finally keyword is to cleanup resources allocated by try block, such as closing file, closing database connection, etc.

- Only one finally block is allowed for each try block.
- Use of finally block is optional.
- **BUILT-IN EXCEPTIONS IN JAVA**

The Java programming language has several built-in exception class that support exception handling. Every exception class is suitable to explain certain error situations at run time.
All the built-in exception classes in Java were defined a package **java.lang**.
Few built-in exceptions in Java are shown in the following image.



List of checked exceptions in Java

The following table shows the list of several checked exceptions.

S. No. Exception Class with Description

1 ClassNotFoundException

It is thrown when the Java Virtual Machine (JVM) tries to load a particular class and the specified class cannot be found in the classpath.

2 CloneNotSupportedException

Used to indicate that the clone method in class Object has been called to clone an object, but that the object's class does not implement the Cloneable interface.

3 IllegalAccessException

It is thrown when one attempts to access a method or member that visibility qualifiers do not allow.

4 InstantiationException

It is thrown when an application tries to create an instance of a class using the newInstance method in class Class , but the specified class object cannot be instantiated because it is an interface or is an abstract class.

5 InterruptedException

It is thrown when a thread that is sleeping, waiting, or is occupied is interrupted.

6 NoSuchFieldException

It indicates that the class doesn't have a field of a specified name.

7 NoSuchMethodException

It is thrown when some JAR file has a different version at runtime that it had at compile time, a NoSuchMethodException occurs during reflection when we try to access a method that does not exist.

List of unchecked exceptions in Java

The following table shows the list of several unchecked exceptions.

S. No. Exception Class with Description

1 ArithmeticException

It handles the arithmetic exceptions like dividion by zero

2 ArrayIndexOutOfBoundsException

It handles the situations like an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

3 ArrayStoreException

It handles the situations like when an attempt has been made to store the wrong type of object into an array of objects

4 AssertionError

It is used to indicate that an assertion has failed

5 ClassCastException

It handles the situation when we try to improperly cast a class from one type to another.

6 IllegalArgumentException

This exception is thrown in order to indicate that a method has been passed an illegal or inappropriate argument.

7 IllegalMonitorStateException

This indicates that the calling thread has attempted to wait on an object's monitor, or has attempted to notify other threads that wait on an object's monitor, without owning the specified monitor.

8 IllegalStateException

It signals that a method has been invoked at an illegal or inappropriate time.

9 IllegalThreadStateException

It is thrown by the Java runtime environment, when the programmer is trying to modify the state of the thread when it is illegal.

10 IndexOutOfBoundsException

It is thrown when attempting to access an invalid index within a collection, such as an array , vector , string , and so forth.

11 NegativeArraySizeException

It is thrown if an applet tries to create an array with negative size.

12 NullPointerException

it is thrown when program attempts to use an object reference that has the null value.

13 NumberFormatException

It is thrown when we try to convert a string into a numeric value such as float or integer, but the format of the input string is not appropriate or illegal.

14 SecurityException

It is thrown by the Java Card Virtual Machine to indicate a security violation.

15 StringIndexOutOfBounds

It is thrown by the methods of the String class, in order to indicate that an index is either negative, or greater than the size of the string itself.

16 UnsupportedOperationException

It is thrown to indicate that the requested operation is not supported.

CUSTOM EXCEPTIONS

The Java programming language allows us to create our own exception classes which are basically subclasses built-in class Exception.

To create our own exception class simply creates a class as a subclass of built-in Exception class.

We may create constructor in the user-defined exception class and pass a string to Exception class constructor using super(). We can use getMessage() method to access the string.

Files and I/O Streams

FILE CLASS:

The **File** is a built-in class in Java. In java, the File class has been defined in the **java.io** package. The File class represents a reference to a file or directory. The File class has various methods to perform operations like creating a file or directory, reading from a file, updating file content, and deleting a file or directory. The File class in java has the following constructors.

S.No.	Constructor with Description
1	**File(String pathname)** It creates a new File instance by converting the givenpathname string into an abstract pathname. If the given string isthe empty string, then the result is the empty abstract pathname.
2	**File(String parent, String child)** It Creates a new File instance from a parent abstractpathname and a child pathname string. If parent is null then the new File instance is created as if by invoking thesingle-argument File constructor on the given child pathname string.
3	**File(File parent, String child)** It creates a new File instance from a parent abstractpathname and a child pathname string. If parent is null then the new File instance is created as if by invoking thesingle-argument File constructor on the given child pathname string.
4	**File(URI uri)** It creates a new File instance by converting the given file: URI into an abstract pathname.

The File class in java has the following methods.

S.No.	Methods with Description
1	**String getName()** It returns the name of the file or directory that referenced by the current File object.
2	**String getParent()** It returns the pathname of the pathname's parent, or null if the pathname does not name a parent directory.
3	**String getPath()** It returns the path of curent File.

4	**File getParentFile()** It returns the path of the current file's parent; or null if it does not exist.
5	**String getAbsolutePath()** It returns the current file or directory path from the root.
6	**boolean isAbsolute()** It returns true if the current file is absolute, false otherwise.
7	**boolean isDirectory()** It returns true, if the current file is a directory; otherwise returns false.
8	**boolean isFile()** It returns true, if the current file is a file; otherwise returns false.
9	**boolean exists()** It returns true if the current file or directory exist; otherwise returns false.
10	**boolean canRead()** It returns true if and only if the file specified exists and can be read by the application; false otherwise.
11	**boolean canWrite()** It returns true if and only if the file specified exists and the application is allowed to write to the file; false otherwise.
12	**long length()** It returns the length of the current file.
13	**long lastModified()** It returns the time that specifies the file was last modified.
14	**boolean createNewFile()** It returns true if the named file does not exist and was successfully created; false if the named file already exists.
15	**boolean delete()** It deletes the file or directory. And returns true if and only if the file or directory is successfully deleted; false otherwise.
16	**void deleteOnExit()** It sends a requests that the file or directory needs be deleted when the virtual machine terminates.
17	**boolean mkdir()** It returns true if and only if the directory was created; false otherwise.
18	**boolean mkdirs()** It returns true if and only if the directory was created, along with all necessary parent directories; false otherwise.
19	**boolean renameTo(File dest)** It renames the current file. And returns true if and only if the renaming succeeded; false otherwise.
20	**boolean setLastModified(long time)** It sets the last-modified time of the file or directory. And returns true if and only if the operation succeeded; false otherwise.
21	**boolean setReadOnly()** It sets the file permission to only read operations; Returns true if and only if the operation succeeded; false otherwise.
22	**String[] list()** It returns an array of strings containing names of all the files and directories in the current directory.

23	**String[] list(FilenameFilter filter)** It returns an array of strings containing names of all the files and directories in the current directory that satisfy the specified filter.
24	**File[] listFiles()** It returns an array of file references containing names of all the files and directories in the current directory.
25	**File[] listFiles(FileFilter filter)** It returns an array of file references containing names of all the files and directories in the current directory that satisfy the specified filter.
26	**boolean equals(Object obj)** It returns true if and only if the argument is not null and is an abstract pathname that denotes the same file or directory as this abstract pathname.
27	**int compareTo(File pathname)** It Compares two abstract pathnames lexicographically. It returns zero if the argument is equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.
28	**int compareTo(File pathname)** Compares this abstract pathname to another object. Returns zero if the argument is equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.

Let's look at the following code to illustrate file operations.

Example

```
import java.io.*;
public class FileClassTest {
    public static void main(String args[]) {
            File f = new File("C:\\Raja\\datFile.txt");
            System.out.println("Executable File : " + f.canExecute());
            System.out.println("Name of the file : " + f.getName());
            System.out.println("Path of the file : " + f.getAbsolutePath());
            System.out.println("Parent name :  " + f.getParent());
            System.out.println("Write mode :  " + f.canWrite());
            System.out.println("Read mode :  " + f.canRead());
            System.out.println("Existance :  " + f.exists());
            System.out.println("Last Modified :  " + f.lastModified());
            System.out.println("Length :  " + f.length());
            //f.createNewFile()
            //f.delete();
            //f.setReadOnly()
    }}
```

When we run the above program, it produce the following output.

Let's look at the following java code to list all the files in a directory including the files present in all its subdirectories.

Example

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

```
public class ListingFiles {
	public static void main(String[] args) {
		String path = null;
		Scanner read = new Scanner(System.in);
		System.out.print("Enter the root directory name: ");
		path = read.next() + ":\\";
		File f_ref = new File(path);
		if (!f_ref.exists()) {
			printLine();
			System.out.println("Root directory does not exists!");
			printLine();
		} else {
			String ch = "y";
			while (ch.equalsIgnoreCase("y")) {
				printFiles(path);
				System.out.print("Do you want to open any sub-directory (Y/N):  ");
				ch = read.next().toLowerCase();
				if (ch.equalsIgnoreCase("y")) {
					System.out.print("Enter the sub-directory name: ");
					path = path + "\\\\" + read.next();
					File f_ref_2 = new File(path);
					if (!f_ref_2.exists()) {
						printLine();
						System.out.println("The sub-directory does not exists!");
						printLine();
						int lastIndex = path.lastIndexOf("\\");
						path = path.substring(0, lastIndex);
					}
				}
			}
		}
		System.out.println("***** Program Closed *****");
	}

	public static void printFiles(String path) {
		System.out.println("Current Location: " + path);
		File f_ref = new File(path);
		File[] filesList = f_ref.listFiles();
		for (File file : filesList) {
			if (file.isFile())
				System.out.println("- " + file.getName());
			else
				System.out.println("> " + file.getName());
		}
	}

	public static void printLine() {
		System.out.println("----------------------------------------");
	}
```

STREAM BASED I/O (JAVA.IO)

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

STREAM

In Java, streams are the sequence of data that are read from the source and written to the destination.

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1. System.in: This is the standard input stream that is used to read characters from the keyboard or any other standard input device.

2. System.out: This is the standard output stream that is used to produce the result of a program on an output device like the computer screen.

3. System.err: This is the standard error stream that is used to output all the error data that a program might throw, on a computer screen or any standard output device.

TYPES OF STREAMS

Depending on the type of operations, streams can be divided into two primary classes: InPutStream − The InputStream is used to read data from a source.

OutPutStream − The OutputStream is used for writing data to a destination.

Depending upon the data a stream can be classified into:

1. Byte Stream
2. Character Stream

1. BYTE STREAM

Java byte streams are used to perform input and output of 8-bit bytes.

Byte Stream Classes

All byte stream classes are derived from base abstract classes called InputStream and OutputStream.

InputStream Class

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

Subclasses of InputStream

In order to use the functionality of InputStream, we can use its subclasses. Some of them are:

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
DataInputStream	Contains method for reading java standard datatype
FileInputStream	Input stream that reads from a file

Methods of InputStream

The InputStream class provides different methods that are implemented by its subclasses. Here are some of the commonly used methods:

- read() - reads one byte of data from the input stream
- read(byte[] array) - reads bytes from the stream and stores in the specified array
- available() - returns the number of bytes available in the input stream
- mark() - marks the position in the input stream up to which data has been read
- reset() - returns the control to the point in the stream where the mark was set
- close() - closes the input stream

OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes.

Subclasses of OutputStream

In order to use the functionality of OutputStream, we can use its subclasses. Some of them are:

Stream class	Description
BufferedOutputStream	Used for Buffered Output Stream.
DataOutputStream	n method for writing java standard data type
FileOutputStream	Output stream that write to a file.
PrintStream	Output Stream that contain print() and println() method

Methods of OutputStream

The OutputStream class provides different methods that are implemented by its subclasses. Here are some of the methods:

- write() - writes the specified byte to the output stream
- write(byte[] array) - writes the bytes from the specified array to the output stream
- flush() - forces to write all data present in output stream to the destination
- close() - closes the output stream

2. CHARACTER STREAM

Character stream is used to read and write a single character of data.

Character Stream Classes

All the character stream classes are derived from base abstract classes Reader and Writer.

Reader Class

The Reader class of the java.io package is an abstract super class that represents a stream of characters.

Sub classes of Reader Class

In order to use the functionality of Reader, we can use its subclasses. Some of them are:

Stream class	Description

BufferedReader	Handles buffered input stream.
FileReader	Input stream that reads from file.
InputStreamReader	Input stream that translate byte to character

Methods of Reader

The Reader class provides different methods that are implemented by its subclasses. Here are Some of the commonly used methods:

- ready() - checks if the reader is ready to be read
- read(char[] array) - reads the characters from the stream and stores in the specified array
- read(char[] array, int start, int length) - reads the number of characters equal to length from the stream and stores in the specified array starting from the start
- mark() - marks the position in the stream up to which data has been read
- reset() - returns the control to the point in the stream where the mark is set
- skip() - discards the specified number of characters from the stream

Writer Class

- The Writer class of the java.io package is an abstract super class that represents a stream of
- characters.
- Since Writer is an abstract class, it is not useful by itself. However, its subclasses can be used to write data.

Subclasses of Writer

Stream class	Description
BufferedWriter	Handles buffered output stream.
FileWriter	Output stream that writes to file.
PrintWriter	Output Stream that contain print() and println() method.

Methods of Writer

The Writer class provides different methods that are implemented by its subclasses. Here are some of the methods:

- write(char[] array) - writes the characters from the specified array to the output stream
- write(String data) - writes the specified string to the writer
- append(char c) - inserts the specified character to the current writer

- flush() - forces to write all the data present in the writer to the corresponding destination
- close() - closes the writer

RANDOM ACCESS FILE IN JAVA

In java, the **java.io** package has a built-in class **RandomAccessFile** that enables a file to be accessed randomly. The RandomAccessFile class has several methods used to move the cursor position in a file. A random access file behaves like a large array of bytes stored in a file.

RandomAccessFile Constructors

The RandomAccessFile class in java has the following constructors.

S.No.	Constructor with Description
1	**RandomAccessFile(File fileName, String mode)** It creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.
2	**RandomAccessFile(String fileName, String mode)** It creates a random access file stream to read from, and optionally to write to, a file with the specified fileName.

Access Modes

Using the RandomAccessFile, a file may created in th following modes.

- **r** - Creates the file with read mode; Calling write methods will result in an IOException.
- **rw** - Creates the file with read and write mode.
- **rwd** - Creates the file with read and write mode - synchronously. All updates to file content is written to the disk synchronously.
- **rws** - Creates the file with read and write mode - synchronously. All updates to file content or meta data is written to the disk synchronously.

RandomAccessFile methods

The RandomAccessFile class in java has the following methods.

S.No.	Methods with Description
1	**int read()** It reads byte of data from a file. The byte is returned as an integer in the range 0-255.
2	**int read(byte[] b)** It reads byte of data from file upto b.length, -1 if end of file is reached.
3	**int read(byte[] b, int offset, int len)** It reads bytes initialising from offset position upto b.length from the buffer.
4	**boolean readBoolean()** It reads a boolean value from from the file.
5	**byte readByte()** It reads signed eight-bit value from file.
6	**char readChar()** It reads a character value from file.

7	**double readDouble()** It reads a double value from file.
8	**float readFloat()** It reads a float value from file.
9	**long readLong()** It reads a long value from file.
10	**int readInt()** It reads a integer value from file.
11	**void readFully(byte[] b)** It reads bytes initialising from offset position upto b.length from the buffer.
12	**void readFully(byte[] b, int offset, int len)** It reads bytes initialising from offset position upto b.length from the buffer.
13	**String readUTF()** t reads in a string from the file.
14	**void seek(long pos)** It sets the file-pointer(cursor) measured from the beginning of the file, at which the next read or write occurs.
15	**long length()** It returns the length of the file.
16	**void write(int b)** It writes the specified byte to the file from the current cursor position.
17	**void writeFloat(float v)** It converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a four-byte quantity, high byte first.
18	**void writeDouble(double v)** It converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an eight-byte quantity, high byte first.

Let's look at the following example program.

Example

```
import java.io.*;
public class RandomAccessFileDemo
{
   public static void main(String[] args)
   {
      try
      {
         double d = 1.5;
         float f = 14.56f;

         // Creating a new RandomAccessFile - "F2"
         RandomAccessFile f_ref = new RandomAccessFile("C:\\Raja\\Input-File.txt", "rw");

         // Writing to file
         f_ref.writeUTF("Hello, Good Morning!");

         // File Pointer at index position - 0
         f_ref.seek(0);
```

```
// read() method :
System.out.println("Use of read() method : " + f_ref.read());

f_ref.seek(0);

byte[] b = {1, 2, 3};

// Use of .read(byte[] b) method :
System.out.println("Use of .read(byte[] b) : " + f_ref.read(b));

// readBoolean() method :
System.out.println("Use of readBoolean() : " + f_ref.readBoolean());

// readByte() method :
System.out.println("Use of readByte() : " + f_ref.readByte());

f_ref.writeChar('c');
f_ref.seek(0);

// readChar() :
System.out.println("Use of readChar() : " + f_ref.readChar());

f_ref.seek(0);
f_ref.writeDouble(d);
f_ref.seek(0);

// read double
System.out.println("Use of readDouble() : " + f_ref.readDouble());

f_ref.seek(0);
f_ref.writeFloat(f);
f_ref.seek(0);

// readFloat() :
System.out.println("Use of readFloat() : " + f_ref.readFloat());

f_ref.seek(0);
// Create array upto geek.length
byte[] arr = new byte[(int) f_ref.length()];
// readFully() :
f_ref.readFully(arr);

String str1 = new String(arr);
System.out.println("Use of readFully() : " + str1);

f_ref.seek(0);

// readFully(byte[] b, int off, int len) :
f_ref.readFully(arr, 0, 8);
```

```
            String str2 = new String(arr);
            System.out.println("Use of readFully(byte[] b, int off, int len) : " + str2);
        }
        catch (IOException ex)
        {
            System.out.println("Something went Wrong");
            ex.printStackTrace();
        }
    }
}
```

UNIT - III

Packages: Defining a Package, CLASSPATH, Access Specifiers, importing packages. Few Utility Classes - String Tokenizer, BitSet, Date, Calendar, Random, Formatter, Scanner.

Collections: Collections overview, Collection Interfaces, Collections Implementation Classes, Sorting in Collections, Comparable and Comparator Interfaces..

PACKAGES:

DEFINING PACKAGES

In java, a package is a container of classes, interfaces, and sub-packages. We may think of it as a folder in a file directory.

We use the packages to avoid naming conflicts and to organize project-related classes, interfaces, and sub-packages into a bundle.

In java, the packages have divided into two types.

- Built-in Packages
- User-defined Packages

Built-in Packages

The built-in packages are the packages from java API. The Java API is a library of pre-defined classes, interfaces, and sub-packages. The built-in packages were included in the JDK. There are many built-in packages in java, few of them are as java, lang, io, util, awt, javax, swing, net, sql, etc. We need to import the built-in packages to use them in our program. To import a package, we use the import statement.

User-defined Packages

The user-defined packages are the packages created by the user. User is free to create their own packages.

Definig a Package in java

We use the ***package*** keyword to create or define a package in java programming language.

Syntax

package packageName;

- The package statement must be the first statement in the program.
- The package name must be a single word.

- The package name must use Camel case notation.

Let's consider the following code to create a user-defined package myPackage.

Example

```
package myPackage;
public class DefiningPackage {
	public static void main(String[] args) {
		System.out.println("This class belongs to myPackage.");
	}
}
```

Now, save the above code in a file **DefiningPackage.java**, and compile it using the following command.

javac -d . DefiningPackage.java

The above command creates a directory with the package name ***myPackage***, and the ***DefiningPackage.class*** is saved into it. Run the program use the following command.

java myPackage.DefiningPackage

ACCESS PROTECTION IN JAVA PACKAGES

In java, the access modifiers define the accessibility of the class and its members. For example, private members are accessible within the same class members only. Java has four access modifiers, and they are default, private, protected, and public.

In java, the package is a container of classes, sub-classes, interfaces, and sub-packages. The class acts as a container of data and methods. So, the access modifier decides the accessibility of class members across the different packages.

In java, the accessibility of the members of a class or interface depends on its access specifiers. The following table provides information about the visibility of both data members and methods.

Access control for members of class and interface in java

Accessibility / Access Specifier / Location	Same Class	Same Package		Other Package	
		Child class	Non-child class	Child class	Non-child class
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

www.btechsmartclass.com

- The **public** members can be accessed everywhere.
- The **private** members can be accessed only inside the same class.
- The **protected** members are accessible to every child class (same package or other packages).
- The **default** members are accessible within the same package but not outside the package.

IMPORTING PACKAGES IN JAVA

In java, the ***import*** keyword used to import built-in and user-defined packages. When a package has imported, we can refer to all the classes of that package using their name directly.
The import statement must be after the package statement, and before any other statement.
Using an import statement, we may import a specific class or all the classes from a package.

- Using one import statement, we may import only one package or a class.
- Using an import statement, we cannot import a class directly, but it must be a part of a package.
- A program may contain any number of import statements.

Importing specific class
Using an importing statement, we can import a specific class. The following syntax is employed to import a specific class.
Syntax
import packageName.ClassName;

Let's look at an import statement to import a built-in package and Scanner class.
Example

```
package myPackage;
import java.util.Scanner;
public class ImportingExample {
	public static void main(String[] args) {
		Scanner read = new Scanner(System.in);
		int i = read.nextInt();
		System.out.println("You have entered a number " + i);
	}
}
```

In the above code, the class **ImportingExample** belongs to **myPackage** package, and it also importing a class called **Scanner** from **java.util** package.

Importing all the classes
Using an importing statement, we can import all the classes of a package. To import all the classes of the package, we use * symbol. The following syntax is employed to import all the classes of a package.
Syntax
import packageName.*;

Let's look at an import statement to import a built-in package.
Example

```
package myPackage;
import java.util.*;
public class ImportingExample {
	public static void main(String[] args) {
		Scanner read = new Scanner(System.in);
```

```
int i = read.nextInt();
System.out.println("You have entered a number " + i);
Random rand = new Random();

int num = rand.nextInt(100);
System.out.println("Randomly generated number " + num);     }}
```

In the above code, the class **ImportingExample** belongs to **myPackage** package, and it also importing all the classes like Scanner, Random, Stack, Vector, ArrayList, HashSet, etc. from the **java.util** package.

- The import statement imports only classes of the package, but not sub-packages and its classes.
- We may also import sub-packages by using a symbol '.' (dot) to separate parent package and sub-package.

Consider the following import statement.
import java.util.*;

The above import statement **util** is a sub-package of **java** package. It imports all the classes of **util** package only, but not classes of **java** package.

STRINGTOKENIZER CLASS

The StringTokenizer is a built-in class in java used to break a string into tokens. The StringTokenizer class is available inside the java.util package.
The StringTokenizer class object internally maintains a current position within the string to be tokenized.
🔔 A token is returned by taking a substring of the string that was used to create the StringTokenizer object.

The StringTokenizer class in java has the following constructor.

S. No.	Constructor with Description
1	**StringTokenizer(String str)** It creates StringTokenizer object for the specified string str with default delimeter.
2	**StringTokenizer(String str, String delimeter)** It creates StringTokenizer object for the specified string str with specified delimeter.
3	**StringTokenizer(String str, String delimeter, boolean returnValue)** It creates StringTokenizer object with specified string, delimeter and returnValue.

The StringTokenizer class in java has the following methods.

S.No.	Methods with Description
1	**String nextToken()** It returns the next token from the StringTokenizer object.
2	**String nextToken(String delimeter)** It returns the next token from the StringTokenizer object based on the delimeter.
3	**Object nextElement()** It returns the next token from the StringTokenizer object.

4	**boolean hasMoreTokens()** It returns true if there are more tokens in the StringTokenizer object. otherwise returns false.
5	**boolean hasMoreElements()** It returns true if there are more tokens in the StringTokenizer object. otherwise returns false.
6	**int countTokens()** It returns total number of tokens in the StringTokenizer object.

BITSET CLASS:

The BitSet is a built-in class in java used to create a dynamic array of bits represented by boolean values.
The BitSet class is available inside the java.util package.
The BitSet array can increase in size as needed. This feature makes the BitSet similar to a Vector of bits.

- The bit values can be accessed by non-negative integers as an index.
- The size of the array is flexible and can grow to accommodate additional bit as needed.
- The default value of the BitSet is boolean false with a representation as 0 (off).
- BitSet uses 1 bit of memory per each boolean value.

The BitSet class in java has the following constructor.

S. No.	Constructor with Description
1	**BitSet( )** It creates a default BitSet object.
2	**BitSet(int noOfBits)** It creates a BitSet object with number of bits that it can hold. All bits are initialized to zero.

The BitSet class in java has the following methods.

S.No.	Methods with Description
1	**void and(BitSet bitSet)** It performs AND operation on the contents of the invoking BitSet object with those specified by bitSet.
2	**void andNot(BitSet bitSet)** For each 1 bit in bitSet, the corresponding bit in the invoking BitSet is cleared.
3	**void flip(int index)** It reverses the bit at specified index.
4	**void flip(int startIndex, int endIndex)** It reverses all the bits from specified startIndex to endIndex.
5	**void or(BitSet bitSet)** It performs OR operation on the contents of the invoking BitSet object with those specified by bitSet.
6	**void xor(BitSet bitSet)** It performs XOR operation on the contents of the invoking BitSet object

	with those specified by bitSet.
7	**int cardinality( )** It returns the number of bits set to true in the invoking BitSet.
8	**void clear( )** It sets all the bits to zeros of the invoking BitSet.
9	**void clear(int index)** It set the bit specified by given index to zero.
10	**void clear(int startIndex, int endIndex)** It sets all the bits from specified startIndex to endIndex to zero.
11	**Object clone( )** It duplicates the invoking BitSet object.
12	**boolean equals(Object bitSet)** It retruns true if both invoking and argumented BitSets are equal, otherwise returns false.
13	**boolean get(int index)** It retruns the present state of the bit at given index in the invoking BitSet.
14	**BitSet get(int startIndex, int endIndex)** It returns a BitSet object that consists all the bits from startIndex to endIndex.
15	**int hashCode( )** It returns the hash code of the invoking BitSet.
16	**boolean intersects(BitSet bitSet)** It returns true if at least one pair of corresponding bits within the invoking object and bitSet are 1.
17	**boolean isEmpty( )** It returns true if all bits in the invoking object are zero, otherwise returns false.
17	**int length( )** It returns the total number of bits in the invoking BitSet.
18	**int nextClearBit(int startIndex)** It returns the index of the next cleared bit, (that is, the next zero bit), starting from the index specified by startIndex.
19	**int nextSetBit(int startIndex)** It returns the index of the next set bit (that is, the next 1 bit), starting from the index specified by startIndex. If no bit is set, -1 is returned.
20	**void set(int index)** It sets the bit specified by index.
21	**void set(int index, boolean value)** It sets the bit specified by index to the value passed in value.
22	**void set(int startIndex, int endIndex)** It sets all the bits from startIndex to endIndex.
23	**void set(int startIndex, int endIndex, boolean value)** It sets all the bits to specified value from startIndex to endIndex.
24	**int size( )** It returns the total number of bits in the invoking BitSet.
25	**String toString( )** It returns the string equivalent of the invoking BitSet object.

DATE CLASS

The **Date** is a built-in class in java used to work with date and time in java. The Date class is available

inside the **java.util** package. The Date class represents the date and time with millisecond precision.
The Date class implements **Serializable**, **Cloneable** and **Comparable** interface.

🔔 Most of the constructors and methods of Date class has been deprecated after Calendar class introduced.

The Date class in java has the following constructor.

S. No.	Constructor with Description
1	**Date( )** It creates a Date object that represents current date and time.
2	**Date(long milliseconds)** It creates a date object for the given milliseconds since January 1, 1970, 00:00:00 GMT.
3	**Date(int year, int month, int date) -** Depricated It creates a date object with the specified year, month, and date.
4	**Date(int year, int month, int date, int hrs, int min) -** Depricated It creates a date object with the specified year, month, date, hours, and minuts.
5	**Date(int year, int month, int date, int hrs, int min, int sec) -** Depricated It creates a date object with the specified year, month, date, hours, minuts and seconds.
5	**Date(String s) -** Depricated It creates a Date object and initializes it so that it represents the date and time indicated by the string s, which is interpreted as if by the parse(java.lang.String) method.

The Date class in java has the following methods.

S.No.	Methods with Description
1	**long getTime()** It returns the time represented by this date object.
2	**boolean after(Date date)** It returns true, if the invoking date is after the argumented date.
3	**boolean before(Date date)** It returns true, if the invoking date is before the argumented date.
4	**Date from(Instant instant)** It returns an instance of Date object from Instant date.
5	**void setTime(long time)** It changes the current date and time to given time.
6	**Object clone( )** It duplicates the invoking Date object.
7	**int compareTo(Date date)** It compares current date with given date.
8	**boolean equals(Date date)** It compares current date with given date for equality.
9	**int hashCode()** It returns the hash code value of the invoking date object.
10	**Instant toInstant()**

	It converts current date into Instant object.
11	**String toString()** It converts this date into Instant object.

CALENDAR CLASS

The **Calendar** is a built-in abstract class in java used to convert date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc. The Calendar class is available inside the **java.util** package.

The Calendar class implements **Serializable**, **Cloneable** and **Comparable** interface.

- As the Calendar class is an abstract class, we can not create an object using it.
- We will use the static method **Calendar.getInstance()** to instantiate and implement a sub-class.

The Calendar class in java has the following methods.

S.No.	Methods with Description
1	**Calendar getInstance()** It returns a calendar using the default time zone and locale.
2	**Date getTime()** It returns a Date object representing the invoking Calendar's time value.
3	**TimeZone getTimeZone()** It returns the time zone object associated with the invoking calendar.
4	**String getCalendarType()** It returns an instance of Date object from Instant date.
5	**int get(int field)** It rerturns the value for the given calendar field.
6	**int getFirstDayOfWeek()** It returns the day of the week in integer form.
7	**int getWeeksInWeekYear()** It retruns the total weeks in week year.
8	**int getWeekYear()** It returns the week year represented by current Calendar.
9	**void add(int field, int amount)** It adds the specified amount of time to the given calendar field.
10	**boolean after (Object when)** It returns true if the time represented by the Calendar is after the time represented by when Object.
11	**boolean before(Object when)** It returns true if the time represented by the Calendar is before the time represented by when Object.
12	**void clear(int field)** It sets the given calendar field value and the time value of this Calendar undefined.
13	**Object clone()** It retruns the copy of the current object.
14	**int compareTo(Calendar anotherCalendar)** It compares and retruns the time values (millisecond offsets) between two calendar object.
15	**void complete()** It sets any unset fields in the calendar fields.
16	**void computeFields()**

	It converts the current UTC time value time[] to calendar field values in fields[].
17	**void computeTime()** It converts the current calendar field values in fields[] to the millisecond time value time.
18	**boolean equals(Object object)** It returns true if both invoking object and argumented object are equal.
19	**int getActualMaximum(int field)** It returns the Maximum possible value of the specified calendar field.
20	**int getActualMinimum(int field)** It returns the Minimum possible value of the specified calendar field.
21	**Set getAvailableCalendarTypes()** It returns a string set of all available calendar type supported by Java Runtime Environment.
22	**Locale[] getAvailableLocales()** It returns an array of all locales available in java runtime environment.
23	**String getDisplayName(int field, int style, Locale locale)** It returns the String representation of the specified calendar field value in a given style, and local.
24	**Map getDisplayNames(int field, int style, Locale locale)** It returns Map representation of the given calendar field value in a given style and local.
25	**int getGreatestMinimum(int field)** It returns the highest minimum value of the specified Calendar field.
26	**int getLeastMaximum(int field)** It returns the highest maximum value of the specified Calendar field.
27	**int getMaximum(int field)** It returns the maximum value of the specified calendar field.
28	**int getMinimalDaysInFirstWeek()** It returns required minimum days in integer form.
29	**int getMinimum(int field)** It returns the minimum value of specified calendar field.
30	**long getTimeInMillis()** It returns the current time in millisecond.
31	**int hashCode()** It returns the hash code of the invoking object.
32	**int internalGet(int field)** It returns the value of the given calendar field.
33	**boolean isLenient()** It returns true if the interpretation mode of this calendar is lenient; false otherwise.
34	**boolean isSet(int field)** If not set then it returns false otherwise true.
35	**boolean isWeekDateSupported()** It returns true if the calendar supports week date. The default value is false.
36	**void roll(int field, boolean up)** It increase or decrease the specified calendar field by one unit without affecting the other field
37	**void set(int field, int value)** It sets the specified calendar field by the specified value.

38	**void setFirstDayOfWeek(int value)** It sets the first day of the week.
39	**void setMinimalDaysInFirstWeek(int value)** It sets the minimal days required in the first week.
40	**void setTime(Date date)** It sets the Time of current calendar object.
41	**void setTimeInMillis(long millis)** It sets the current time in millisecond.
42	**void setTimeZone(TimeZone value)** It sets the TimeZone with passed TimeZone value.
43	**void setWeekDate(int weekYear, int weekOfYear, int dayOfWeek)** It sets the current date with specified integer value.
44	**Instant toInstant()** It converts the current object to an instant.
45	**String toString()** It returns a string representation of the current object.

RANDOM CLASS

The **Random** is a built-in class in java used to generate a stream of pseudo-random numbers in java programming. The Random class is available inside the **java.util** package.

The Random class implements **Serializable**, **Cloneable** and **Comparable** interface.

- The **Random** class is a part of java.util package.
- The **Random** class provides several methods to generate random numbers of type integer, double, long, float etc.
- The **Random** class is thread-safe.
- Random number generation algorithm works on the seed value. If not provided, seed value is created from system nano time.

The Random class in java has the following constructors.

S.No.	Constructor with Description
1	**Random()** It creates a new random number generator.
2	**Random(long seedValue)** It creates a new random number generator using a single long seedValue.

The Random class in java has the following methods.

S.No.	Methods with Description
1	**int next(int bits)** It generates the next pseudo-random number.
2	**Boolean nextBoolean()** It generates the next uniformly distributed pseudo-random boolean value.
3	**double nextDouble()** It generates the next pseudo-random double number between 0.0 and 1.0.
4	**void nextBytes(byte[] bytes)** It places the generated random bytes into an user-supplied byte array.
5	**float nextFloat()** It generates the next pseudo-random float number between 0.0 and 1.0..
6	**int nextInt()**

	It generates the next pseudo-random integer number.
7	**int nextInt(int n)** It generates the next pseudo-random integer value between zero and n.
8	**long nextLong()** It generates the next pseudo-random, uniformly distributed long value.
9	**double nextGaussian()** It generates the next pseudo-random Gaussian distributed double number with mean 0.0 and standard deviation 1.0.
10	**void setSeed(long seedValue)** It sets the seed of the random number generator using a single long seedValue.
11	**DoubleStream doubles()** It returns a stream of pseudo-random double values, each conforming between 0.0 and 1.0.
12	**DoubleStream doubles(double start, double end)** It retruns an unlimited stream of pseudo-random double values, each conforming to the given start and end.
13	**DoubleStream doubles(long streamSize)** It returns a stream producing the pseudo-random double values for the given streamSize number, each between 0.0 and 1.0.
14	**DoubleStream doubles(long streamSize, double start, double end)** It returns a stream producing the given streamSizenumber of pseudo-random double values, each conforming to the given start and end.
15	**IntStream ints()** It returns a stream of pseudo-random integer values.
16	**IntStream ints(int start, int end)** It retruns an unlimited stream of pseudo-random integer values, each conforming to the given start and end.
17	**IntStream ints(long streamSize)** It returns a stream producing the pseudo-random integer values for the given streamSize number.
18	**IntStream ints(long streamSize, int start, int end)** It returns a stream producing the given streamSizenumber of pseudo-random integer values, each conforming to the given start and end.
19	**LongStream longs()** It returns a stream of pseudo-random long values.
20	**LongStream longs(long start, long end)** It retruns an unlimited stream of pseudo-random long values, each conforming to the given start and end.
21	**LongStream longs(long streamSize)** It returns a stream producing the pseudo-random long values for the given streamSize number.
22	**LongStream longs(long streamSize, long start, long end)** It returns a stream producing the given streamSizenumber of pseudo-random long values, each conforming to the given start and end.

FORMATTER CLASS

The **Formatter** is a built-in class in java used for layout justification and alignment, common formats for numeric, string, and date/time data, and locale-specific output in java programming. The Formatter class is defined as final class inside the **java.util** package.

The Formatter class implements Closeable and Flushable interface.

The Formatter class in java has the following constructors.

S.No.	Constructor with Description
1	**Formatter()** It creates a new formatter.
2	**Formatter(Appendable a)** It creates a new formatter with the specified destination.
3	**Formatter(Appendable a, Locale l)** It creates a new formatter with the specified destination and locale.
4	**Formatter(File file)** It creates a new formatter with the specified file.
5	**Formatter(File file, String charset)** It creates a new formatter with the specified file and charset.
6	**Formatter(File file, String charset, Locale l)** It creates a new formatter with the specified file, charset, and locale.
7	**Formatter(Locale l)** It creates a new formatter with the specified locale.
8	**Formatter(OutputStream os)** It creates a new formatter with the specified output stream.
9	**Formatter(OutputStream os, String charset)** It creates a new formatter with the specified output stream and charset.
10	**Formatter(OutputStream os, String charset, Locale l)** It creates a new formatter with the specified output stream, charset, and locale.
11	**Formatter(PrintStream ps)** It creates a new formatter with the specified print stream.
12	**Formatter(String fileName)** It creates a new formatter with the specified file name.
13	**Formatter(String fileName, String charset)** It creates a new formatter with the specified file name and charset.
14	**Formatter(String fileName, String charset, Locale l)** It creates a new formatter with the specified file name, charset, and locale.

The Formatter class in java has the following methods.

S.No.	Methods with Description
1	**Formatter format(Locale l, String format, Object... args)** It writes a formatted string to the invoking object's destination using the specified locale, format string, and arguments.
2	**Formatter format(String format, Object... args)** It writes a formatted string to the invoking object's destination using the specified format string and arguments.
3	**void flush()** It flushes the invoking formatter.
4	**Appendable out()** It returns the destination for the output.
5	**Locale locale()** It returns the locale set by the construction of the invoking formatter.

6	**String toString()** It converts the invoking object to string.
7	**IOException ioException()** It returns the IOException last thrown by the invoking formatter's Appendable.
8	**void close()** It closes the invoking formatter.

SCANNER CLASS

The **Scanner** is a built-in class in java used for read the input from the user in java programming. The Scanner class is defined inside the **java.util** package.

The Scanner class implements **Iterator** interface.

The Scanner class provides the easiest way to read input in a Java program.

🔔 The Scanner object breaks its input into tokens using a delimiter pattern, the default delimiter is whitespace.

The Scanner class in java has the following constructors.

S.No.	Constructor with Description
1	**Scanner(InputStream source)** It creates a new Scanner that produces values read from the specified input stream.
2	**Scanner(InputStream source, String charsetName)** It creates a new Scanner that produces values read from the specified input stream.
3	**Scanner(File source)** It creates a new Scanner that produces values scanned from the specified file.
4	**Scanner(File source, String charsetName)** It creates a new Scanner that produces values scanned from the specified file.
5	**Scanner(String source)** It creates a new Scanner that produces values scanned from the specified string.
6	**Scanner(Readable source)** It creates a new Scanner that produces values scanned from the specified source.
7	**Scanner(ReadableByteChannel source)** It creates a new Scanner that produces values scanned from the specified channel.
8	**Scanner(ReadableByteChannel source, String charsetName)** It creates a new Scanner that produces values scanned from the specified channel.

The Scanner class in java has the following methods.

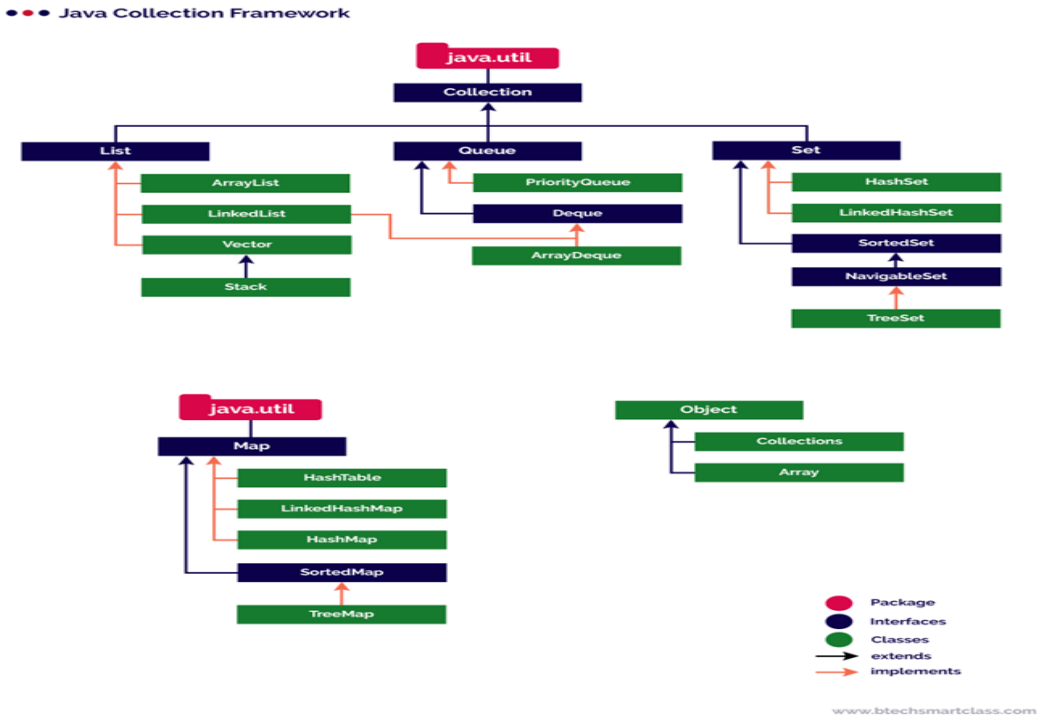
S.No.	Methods with Description
1	**String next()** It reads the next complete token from the invoking scanner.
2	**String next(Pattern pattern)** It reads the next token if it matches the specified pattern.
3	**String next(String pattern)**

	It reads the next token if it matches the pattern constructed from the specified string.
4	**boolean nextBoolean()** It reads a boolean value from the user.
5	**byte nextByte()** It reads a byte value from the user.
6	**double nextDouble()** It reads a double value from the user.
7	**float nextFloat()** It reads a floating-point value from the user.
8	**int nextInt()** It reads an integer value from the user.
9	**long nextLong()** It reads a long value from the user.
10	**short nextShort()** It reads a short value from the user.
11	**String nextLine()** It reads a string value from the user.
12	**boolean hasNext()** It returns true if the invoking scanner has another token in its input.
13	**void remove()** It is used when remove operation is not supported by this implementation of Iterator.
14	**void close()** It closes the invoking scanner.

Collections:

Java Collection Framework

Java collection framework is a collection of interfaces and classes used to storing and processing a group of individual objects as a single unit. The java collection framework holds several classes that provide a large number of methods to store and process a group of objects. These classes make the programmer task super easy and fast.
Java collection framework was introduced in java 1.2 version.
Java collection framework has the following hierarchy.



Collection Interfaces:

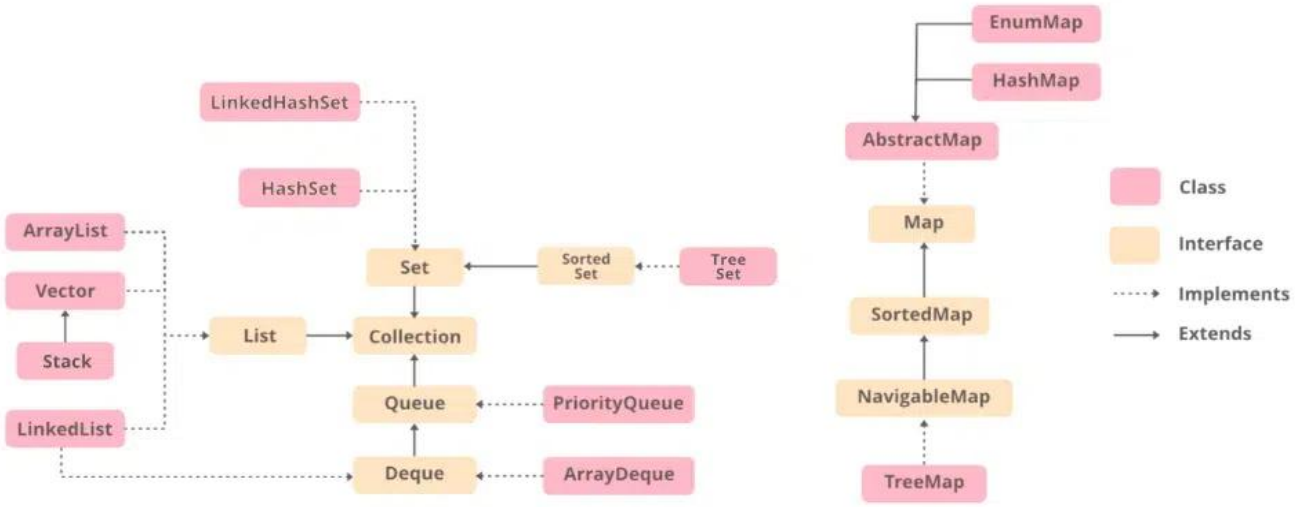
The Collection interface is the root interface for most of the interfaces and classes of collection framework. The Collection interface is available inside the java.util package. It defines the methods that are commonly used by almost all the collections.

The Collection interface extends **Iterable** interface.

Collections Implementation Classes:

Collections class in Java is one of the utility classes in Java Collections Framework. The java.util package contains

the Collections class in Java. Java Collections class is used with the static methods that operate on the collections or return the collection. All the methods of this class throw the NullPointerException if the collection or object passed to the methods is null.



1. ArrayList

ArrayList is a class implemented using a list interface, in that provides the functionality of a dynamic array where the size of the array is not fixed.

Syntax:

ArrayList<_type_> var_name = new ArrayList<_type_>();

2. Vector

Vector is a Part of the collection class that implements a dynamic array that can grow or shrink its size as required.

Syntax:

public class Vector<E> extends AbstractList<E> implements List<E>, RandomAccess,

Cloneable, Serializable

3. Stack

Stack is a part of Java collection class that models and implements a Stack data structure. It is based on the basic principle of last-in-first-out(LIFO).

Syntax:

public class Stack<E> extends Vector<E>

4. LinkedList

LinkedList class is an implementation of the LinkedList data structure. It can store the elements that are not stored in contiguous locations and every element is a separate object with a different data part and different address part.

Syntax:

LinkedList<_type_> var_name = new LinkedList<_type_>();

5. HashSet

HashSet is implemented using the Hashtable data structure. It offers constant time performance for the performing operations like add, remove, contains, and size.

Syntax:

public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable

6. LinkedHashSet

LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked List across all elements.

Syntax:

public class LinkedHashSet<E> extends HashSet<E> implements Set<E>, Cloneable, Serializable

7. TreeSet

TreeSet class is implementations of the SortedSet interface in Java that uses a Tree for storage. The ordering of the elements is maintained by a set using their natural ordering whether an explicit comparator is provided or not.

Syntax:

TreeSet<E> set = new TreeSet<>();

8. PriorityQueue

The PriorityQueue is based on the priority heap. The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.

Syntax:

public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable

9. ArrayDeque

The ArrayDeque class in Java is an implementation of the Deque interface that uses a resizable array to store its elements. The ArrayDeque class provides constant-time performance for inserting and removing elements from both ends.

Syntax:

public class ArrayDeque<E> extends

AbstractCollection<E> implements Deque<E>, Cloneable,Serializable

10. HashMap

HashMap Class is similar to HashTable but the data unsynchronized. It stores the data in (Key, Value) pairs, and you can access them by an index of another type.

Syntax:

public class HashMap<K,V> extends AbstractMap<K,V>

implements Map<K,V>, Cloneable, Serializable

11. EnumMap

EnumMap extends AbstractMap and implements the Map interface in Java.

Syntax:

public class EnumMap<K extends Enum<K>,V> extends

AbstractMap<K,V> implements Serializable, Cloneable

12. AbstractMap

The AbstractMap class is a part of the Java Collection Framework. It implements the Map interface to provide a structure to it, by doing so it makes the further implementations easier.

Syntax:

public abstract class AbstractMap<K,V> implements Map<K,V>

13. TreeMap

A TreeMap is implemented using a Red-Black tree.TreeMap provides an ordered collection of key-value pairs, where the keys are ordered based on their natural order or a custom Comparator passed to the constructor.

Syntax:

SortedMap<K, V> m = Collections.synchronizedSortedMap(new TreeMap<>());

Sorting in Collections:

java.util.Collections.sort() method is present in java.util.Collections class. It is used to sort the elements present in the specified list of Collection in ascending order. It works similar to java.util.Arrays.sort() method but it is better than as it can sort the elements of Array as well as linked list, queue and many more present in it.

Comparable and Comparator Interfaces:

Comparable and Comparator both are interfaces and can be used to sort collection elements.

Comparable	Comparator
1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable **affects the original class**, i.e., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo() method** to	Comparator provides **compare()**

sort elements.	**method** to sort elements.
4) Comparable is present in **java.lang** package.	A Comparator is present in the **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.

Unit-IV

Multithreading: Process and Thread, Differences between thread-based multitasking and process based multitasking, Java thread life cycle, creating threads, thread priorities, synchronizing threads, inter thread communication. Java Database Connectivity: Types of Drivers, JDBC architecture, JDBC Classes and Interfaces, Basic steps in Developing JDBC Application, Creating a New Database and Table with JDBC.

MULTITHREADING:

Process: Running Program is called process**.**

Thread:

The java programming language allows us to create a program that contains one or more parts that can run simultaneously at the same time. This type of program is known as a multithreading program. Each part of this program is called a thread. Every thread defines a separate path of execution in java. A thread is explained in different ways, and a few of them are as specified below.

A thread is a light wieght process.

A thread may also be defined as follows.

A thread is a subpart of a process that can run individually.

In java, multiple threads can run at a time, which enables the java to write multitasking programs. The multithreading is a specialized form of multitasking. All modern operating systems support multitasking. There are two types of multitasking, and they are as follows.

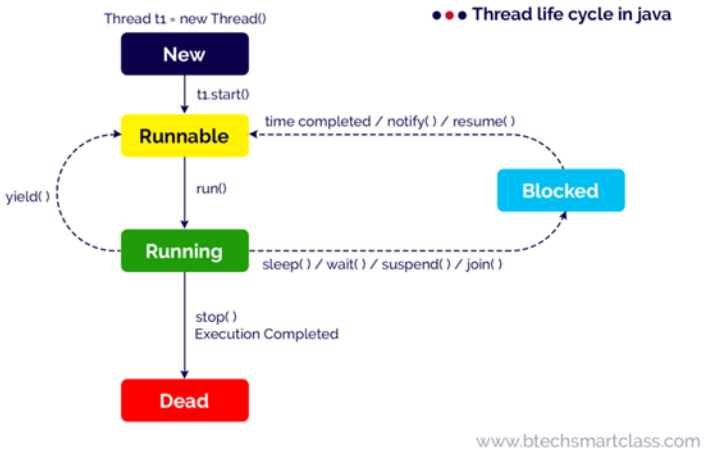
- Process-based multitasking
- Thread-based multitasking

It is important to know the difference between process-based and thread-based multitasking

Differences between thread-based multitasking and process based multitasking

Process-based multitasking	**Thread-based multitasking**
It allows the computer to run two or more programs concurrently	It allows the computer to run two or more threads concurrently
In this process is the smallest unit.	In this thread is the smallest unit.
Process is a larger unit.	Thread is a part of process.
Process is heavy weight.	Thread is light weight.
Process requires seperate address space for each.	Threads share same address space.
Process never gain access over idle time of CPU.	Thread gain access over idle time of CPU.
Inter process communication is expensive.	Inter thread communication is not expensive.

Java thread life cycle



New
When a thread object is created using new, then the thread is said to be in the New state. This state is also known as Born state.
Example
Thread t1 = new Thread();

Runnable / Ready
When a thread calls start( ) method, then the thread is said to be in the Runnable state. This state is also known as a Ready state.
Example
t1.start( );

Running
When a thread calls run( ) method, then the thread is said to be Running. The run( ) method of a thread called automatically by the start( ) method.

Blocked / Waiting
A thread in the Running state may move into the blocked state due to various reasons like sleep( ) method called, wait( ) method called, suspend( ) method called, and join( ) method called, etc.
When a thread is in the blocked or waiting state, it may move to Runnable state due to reasons like sleep time completed, waiting time completed, notify( ) or notifyAll( ) method called, resume( ) method called, etc.
Example
Thread.sleep(1000);
wait(1000);
wait();
suspened();
notify();
notifyAll();
resume();

Dead / Terminated

A thread in the Running state may move into the dead state due to either its execution completed or the stop( ) method called. The dead state is also known as the terminated state.

CREATING THREADS IN JAVA

In java, a thread is a lightweight process. Every java program executes by a thread called the main thread. When a java program gets executed, the main thread created automatically. All other threads called from the main thread.

The java programming language provides two methods to create threads, and they are listed below.

- **Using Thread class (by extending Thread class)**
- **Uisng Runnable interface (by implementing Runnable interface)**

Let's see how to create threads using each of the above.

Extending Thread class

The java contains a built-in class Thread inside the java.lang package. The Thread class contains all the methods that are related to the threads.

To create a thread using Thread class, follow the step given below.

- **Step-1**: Create a class as a child of Thread class. That means, create a class that extends Thread class.
- **Step-2**: Override the run( ) method with the code that is to be executed by the thread. The run( ) method must be public while overriding.
- **Step-3**: Create the object of the newly created class in the main( ) method.
- **Step-4**: Call the start( ) method on the object created in the above step.

Implementng Runnable interface

The java contains a built-in interface Runnable inside the java.lang package. The Runnable interface implemented by the Thread class that contains all the methods that are related to the threads.

To create a thread using Runnable interface, follow the step given below.

- **Step-1**: Create a class that implements Runnable interface.
- **Step-2**: Override the run( ) method with the code that is to be executed by the thread. The run( ) method must be public while overriding.
- **Step-3**: Create the object of the newly created class in the main( ) method.
- **Step-4**: Create the Thread class object by passing above created object as parameter to the Thread class constructor.
- **Step-5**: Call the start( ) method on the Thread class object created in the above step.

The Thread class has the following consructors.

- **Thread( )**
- **Thread(String threadName)**
- **Thread(Runnable objectName)**
- **Thread(Runnable objectName, String threadName)**

The Thread classs contains the following methods.

Method	Description	Return Value
run( )	Defines actual task of the thread.	void
start( )	It moves thre thread from Ready state to Running state by calling run( ) method.	void
setName(String)	Assigns a name to the thread.	void
getName( )	Returns the name of the thread.	String
setPriority(int)	Assigns priority to the thread.	void
getPriority( )	Returns the priority of the thread.	int
getId( )	Returns the ID of the thread.	long
activeCount()	Returns total number of thread under active.	int
currentThread( )	Returns the reference of the thread that currently in running state.	void
sleep(long)	moves the thread to blocked state till the specified number of milliseconds.	void
isAlive( )	Tests if the thread is alive.	boolean
yield( )	Tells to the scheduler that the current thread is willing to yield its current use of a processor.	void
join( )	Waits for the thread to end.	void

The Thread class in java also contains methods like **stop( )**, **destroy( )**, **suspend( )**, and **resume( )**. But they are depricated.

THREAD PRIORITY

In a java programming language, every thread has a property called priority. Most of the scheduling algorithms use the thread priority to schedule the execution sequence. In java, the thread priority range from 1 to 10. Priority 1 is considered as the lowest priority, and priority 10 is considered as the highest priority. The thread with more priority allocates the processor first.

The java programming language Thread class provides two methods **setPriority(int)**, and **getPriority( )** to

handle thread priorities.

The Thread class also contains three constants that are used to set the thread priority, and they are listed below.

- ***MAX_PRIORITY*** - It has the value 10 and indicates highest priority.
- ***NORM_PRIORITY*** - It has the value 5 and indicates normal priority.
- ***MIN_PRIORITY*** - It has the value 1 and indicates lowest priority.

🔔 The default priority of any thread is 5 (i.e. NORM_PRIORITY).

setPriority( ) method

The setPriority( ) method of Thread class used to set the priority of a thread. It takes an integer range from 1 to 10 as an argument and returns nothing (void).

The regular use of the setPriority( ) method is as follows.

Example

```
threadObject.setPriority(4);
or
threadObject.setPriority(MAX_PRIORITY);
```

getPriority( ) method

The getPriority( ) method of Thread class used to access the priority of a thread. It does not takes anyargument and returns name of the thread as String.

The regular use of the getPriority( ) method is as follows.

Example

```
String threadName = threadObject.getPriority();
```

THREAD SYNCHRONISATION

The java programming language supports multithreading. The problem of shared resources occurs when two or more threads get execute at the same time. In such a situation, we need some way to ensure that the shared resource will be accessed by only one thread at a time, and this is performed by using the concept called synchronization.

🔔 The synchronization is the process of allowing only one thread to access a shared resource at a time.

In java, the synchronization is achieved using the following concepts.

- Mutual Exclusion
- Inter thread communication

In this tutorial, we discuss mutual exclusion only, and the interthread communication will be discussed in the next tutorial.

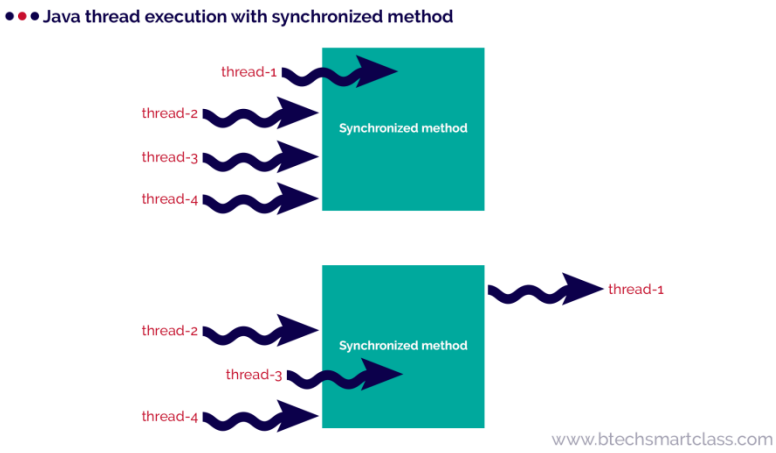
Mutual Exclusion

Using the mutual exclusion process, we keep threads from interfering with one another while they accessing the shared resource. In java, mutual exclusion is achieved using the following concepts.

- Synchronized method
- Synchronized block

Synchronized method

When a method created using a synchronized keyword, it allows only one object to access it at a time. When an object calls a synchronized method, it put a lock on that method so that other objects or thread that are trying to call the same method must wait, until the lock is released. Once the lock is released on the shared resource, one of the threads among the waiting threads will be allocated to the shared resource.



In the above image, initially the thread-1 is accessing the synchronized method and other threads (thread-2, thread-3, and thread-4) are waiting for the resource (synchronized method). When thread-1 completes it task, then one of the threads that are waiting is allocated with the synchronized method, in the above it is thread-3.

Example

```
class Table{
	synchronized void printTable(int n) {
		for(int i = 1; i <= 10; i++)
			System.out.println(n + " * " + i + " = " + i*n);
	}
}

class MyThread_1 extends Thread{
	Table table = new Table();
	int number;
	MyThread_1(Table table, int number){
		this.table = table;
		this.number = number;
	}
	public void run() {
		table.printTable(number);
	}
}

class MyThread_2 extends Thread{

	Table table = new Table();
	int number;
```

```
    MyThread_2(Table table, int number) {
        this.table = table;
        this.number = number;
    }
    public void run() {
        table.printTable(number);
    }
}
public class ThreadSynchronizationExample {

    public static void main(String[] args) {
        Table table = new Table();
        MyThread_1 thread_1 = new MyThread_1(table, 5);
        MyThread_2 thread_2 = new MyThread_2(table, 10);
        thread_1.start();
        thread_2.start();
    }
}
```

Synchronized block

The synchronized block is used when we want to synchronize only a specific sequence of lines in a method. For example, let's consider a method with 20 lines of code where we want to synchronize only a sequence of 5 lines code, we use the synchronized block.

The folllowing syntax is used to define a synchronized block.

Syntax

```
synchronized(object){
   ...
   block code
   ...
}
```

INTER THREAD COMMUNICATION

Inter thread communication is the concept where two or more threads communicate to solve the problem of **polling**. In java, polling is the situation to check some condition repeatedly, to take appropriate action, once the condition is true. That means, in inter-thread communication, a thread waits until a condition becomes true such that other threads can execute its task. The inter-thread communication allows the synchronized threads to communicate with each other.

Java provides the following methods to achieve inter thread communication.

- wait( )
- notify( )
- notifyAll( )

The following table gives detailed description about the above methods.

Method	Description
void wait( )	It makes the current thread to pause its execution until other thread in the same monitor calls notify( )
void notify( )	It wakes up the thread that called wait( ) on the same object.
void notifyAll()	It wakes up all the threads that called wait( ) on the same object.

🔔 Calling notify( ) or notifyAll( ) does not actually give up a lock on a resource.

Let's look at an example problem of producer and consumer. The producer produces the item and the consumer consumes the same. But here, the consumer can not consume until the producer produces the item, and producer can not produce until the consumer consumes the item that already been produced. So here, the consumer has to wait until the producer produces the item, and the producer also needs to wait until the consumer consumes the same. Here we use the inter-thread communication to implement the producer and consumer problem.

The sample implementation of producer and consumer problem is as follows.

Example

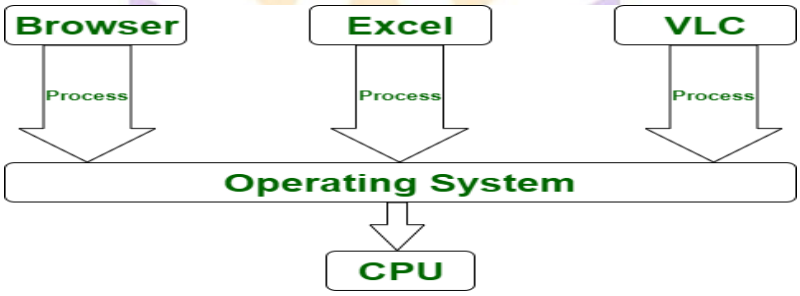
```
class ItemQueue {
    int item;
    boolean valueSet = false;

    synchronized int getItem()

    {
        while (!valueSet)
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
        System.out.println("Consummed:" + item);
        valueSet = false;
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("InterruptedException caught");
        }
        notify();
        return item;
    }

    synchronized void putItem(int item) {
        while (valueSet)
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
        this.item = item;
        valueSet = true;
        System.out.println("Produced: " + item);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("InterruptedException caught");
```

```
            }
            notify();
        }
}

class Producer implements Runnable{
        ItemQueue itemQueue;
        Producer(ItemQueue itemQueue){
                this.itemQueue = itemQueue;
                new Thread(this, "Producer").start();
        }
        public void run() {
                int i = 0;
                while(true) {
                        itemQueue.putItem(i++);
                }
        }
}
class Consumer implements Runnable{

        ItemQueue itemQueue;
        Consumer(ItemQueue itemQueue){
                this.itemQueue = itemQueue;
                new Thread(this, "Consumer").start();
        }
        public void run() {
                while(true) {
                        itemQueue.getItem();
                }
        }
}

class ProducerConsumer{
        public static void main(String args[]) {
                ItemQueue itemQueue = new ItemQueue();
                new Producer(itemQueue);
                new Consumer(itemQueue);

        }
}
```

All the methods wait( ), notify( ), and notifyAll( ) can be used only inside the synchronized methods only.

MULTI-TASKING AND MULTI-THREADING

- Multi-tasking and multi-threading are two techniques used in operating systems to manage multiple processes and tasks.
- Multi-tasking is the ability of an operating system to run multiple processes or tasks concurrently, sharing the same processor and other resources.
- In multi-tasking, the operating system divides the CPU time between multiple tasks, allowing them to execute simultaneously.

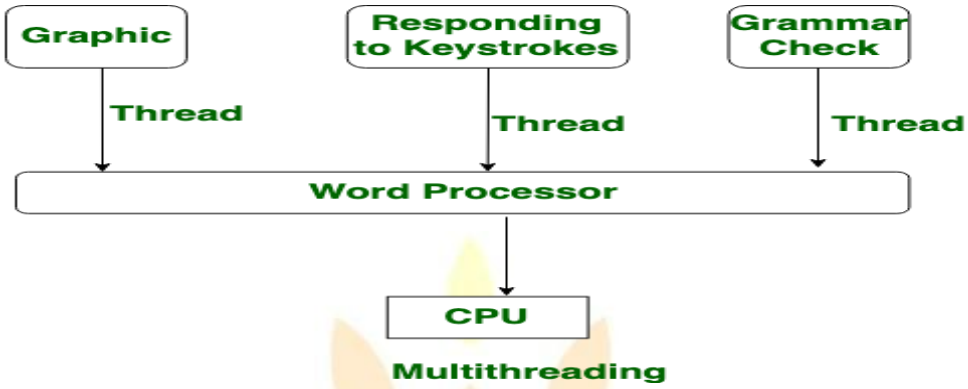


Multitasking

- Each task is assigned a time slice, or a portion of CPU time, during which it can execute its code.
- Multi-tasking is essential for increasing system efficiency, improving user productivity, and achieving optimal resource utilization.

Multi-threading is a technique in which an operating system divides a single process into multiple threads, each of which can execute concurrently.

- Threads share the same memory space and resources of the parent process, allowing them to communicate and synchronize data easily.
- Multi-threading is useful for improving application performance by allowing different parts of the application to execute simultaneously.



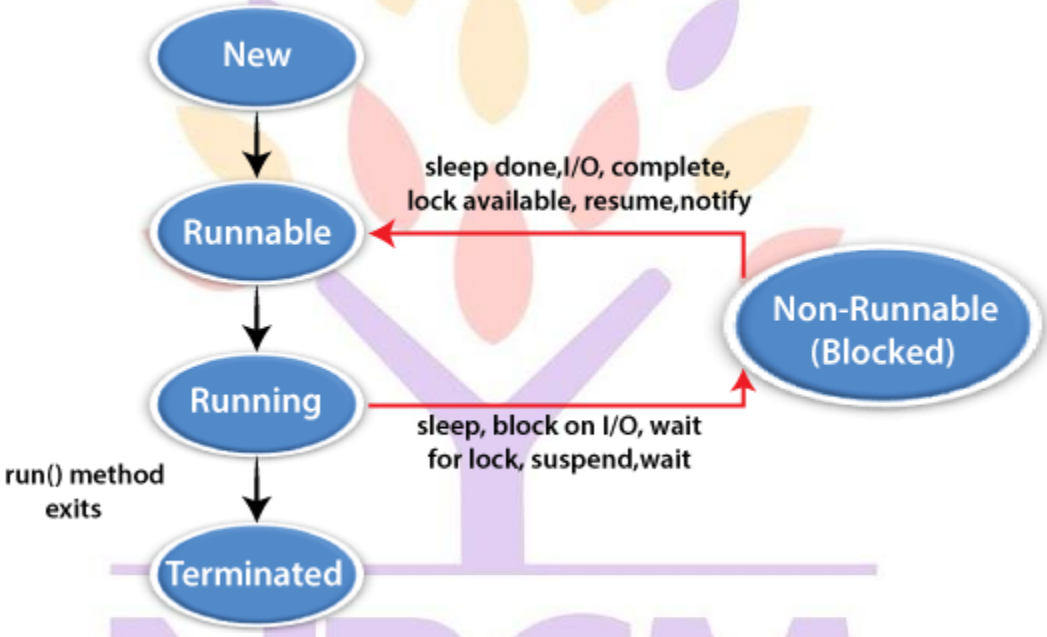
DIFFERENCE BETWEEN MULTI-TASKING AND MULTI-THREADING

Process-based Multitasking	Thread-based Multitasking
Two or more processes/programs can be run concurrently.	Two or more threads can be run concurrently.
The smallest unit of execution is a process.	The smallest unit of execution is a thread.
Inter-process communication is costly and inefficient.	Inter-thread communication is inexpensive and efficient.
Processes take more time for context switching.	Threads take less time for context switching.
It is unable to gain access over CPU idle time.	It can gain access over CPU idle time.
Every program has its own address space.	Threads share the same address space.
Overhead is more.	Overhead is less.

JAVA THREAD MODEL (LIFE CYCLE OF A THREAD)

- In java, a thread goes through different states throughout its execution.
- These stages are called thread life cycle states or phases.
- 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



1. New

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

   Example

   Thread t1 = new Thread();

2. Runnable

- When a thread calls start( ) method, then the thread is said to be in the Runnable state.
- This state is also known as a Ready state.

Example

t1.start( );

3. Running

When a thread calls run( ) method, then the thread is said to be Running. The run( ) method of a thread called automatically by the start( ) method.

4. Non-Runnable (Blocked)

- This is the state when the thread is still alive, but is currently not eligible to run.
- A thread in the Running state may move into the blocked state due to various reasons like sleep( )method called, wait( ) method called, suspend( ) method called, and join( ) method called, etc.
- When a thread is in the blocked or waiting state, it may move to Runnable state due to reasons like sleep time completed, waiting time completed, notify( ) or notifyAll( ) method called, resume( ) method called, etc.

Example

Thread.sleep(1000); wait(1000);

wait(); suspend(); notify(); notifyAll(); resume();

5. Terminated

- A thread in the Running state may move into the dead state due to either its execution completed or the stop( ) method called.
- The dead state is also known as the terminated state.

CREATING THREADS

There are two ways to create a thread:

1. By extending Thread class

2. By implementing Runnable interface.

1. By extending Thread class

The java contains a built-in class Thread inside the java.lang package. The Thread class contains all the methods that are related to the threads.

To create a thread using Thread class, follow the step given below.

Step-1: Create a class as a child of Thread class. That means, create a class that extends Thread class.

Step-2: Override the run( ) method with the code that is to be executed by the thread. The run( ) method must be public while overriding.

Step-3: Create the object of the newly created class in the main( ) method. Step-4: Call the start( ) method on the object created in the above step.

Example: By extending Thread class

```
class SampleThread extends Thread
{
public void run()
{
System.out.println("Thread is under Running..."); for(int i= 1; i<=10; i++)
{
System.out.println("i = " + i);
}
}
}
public class My_Thread_Test
{
```

```
public static void main(String[] args)
{
SampleThread t1 = new SampleThread(); System.out.println("Thread about to start..."); t1.start();
}
}
```

Output: Thread about to start...

Thread is under Running... i = 1 i = 2

i = 3

i = 4

i = 5

i = 6

i = 7

i = 8

i = 9

i = 10

2. By implementing Runnable interface

- The java contains a built-in interface Runnable inside the java.lang package.
- The Runnable interface implemented by the Thread class that contains all the methods that are related to the threads.

To create a thread using Runnable interface, follow the step given below. Step-1: Create a class that implements Runnable interface.

Step-2: Override the run( ) method with the code that is to be executed by the thread. The run( )method must be public while overriding.

Step-3: Create the object of the newly created class in the main( ) method.

Step-4: Create the Thread class object by passing above created object as parameter to the Thread class constructor.

Step-5: Call the start( ) method on the Thread class object created in the above step.

Example: By implementing the Runnable interface

```
class SampleThread implements Runnable
{
public void run()
{
System.out.println("Thread is under Running..."); for(int i= 1; i<=10; i++)
{
System.out.println("i = " + i);
}
}
}
public class My_Thread_Test
{
public static void main(String[] args)
{
SampleThread threadObject = new SampleThread(); Thread thread = new Thread(threadObject);
System.out.println("Thread about to start..."); thread.start();
}
```

}

Output:

Thread about to start...

Thread is under Running... i = 1 i = 2

i = 3

i = 4

i = 5

i = 6

i = 7

i = 8

i = 9

i = 10

CONSTRUCTORS OF THREAD CLASS

1. Thread()
2. Thread(String name)
3. Thread(Runnable r)
4. Thread(Runnable r,String name)

METHODS OF THREAD CLASS

1. public void run(): is used to defines actual task of the thread.
2. public void start():It moves the thread from Ready state to Running state by calling run( ) method.
3. public void sleep(long milliseconds): Moves the thread to blocked state till the specified number of milliseconds.
4. public void join(): waits for a thread to die.

5. public void join(long milliseconds): waits for a thread to die for the specified milliseconds.
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 suspend(): is used to suspend the thread(deprecated).

SLEEP() & JOIN()

```
class SampleThread extends Thread
{
public void run()
{
System.out.println("Thread is under Running..."); for(int i= 1; i<=10; i++)
{
try
{
Thread.sleep(1000);
}
catch(Exception e)
{
```

```
System.out.println(e);
}
System.out.println("i = " + i);
}
}
}
public class My_Thread_Test
{
public static void main(String[] args)
{
SampleThread t1 = new SampleThread(); SampleThread t2 = new SampleThread(); SampleThread t3 =
new SampleThread(); System.out.println("Thread about to start..."); t1.start();
try
{
t1.join();
}
catch(Exception e)
{
System.out.println(e);
}
t2.start();
```

```
t3.start();
}
}
```

THREAD PRIORITIES

- In a java programming language, every thread has a property called 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).
- The thread with more priority allocates the processor first.
- But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

Three constants defined in Thread class:

1. MIN_PRIORITY
2. NORM_PRIORITY
3. 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.
- The java programming language Thread class provides two methods setPriority(int), and getPriority( ) to handle thread priorities.

setPriority( ) method

The setPriority( ) method of Thread class used to set the priority of a thread.

It takes an integer range from 1 to 10 as an argument and returns nothing (void).

Example

```
threadObject.setPriority(4); or
threadObject.setPriority(MAX_PRIORITY);
```

getPriority( ) method

The getPriority( ) method of Thread class used to access the priority of a thread.

It does not takes any argument and returns name of the thread as String.

Example

String threadName = threadObject.getPriority();

Example1

```
class SampleThread extends Thread
{
public void run()
{
System.out.println("Inside SampleThread"); System.out.println("CurrentThread: " +
Thread.currentThread().getName());
}
}
public class My_Thread_Test { public static void main(String[] args)
{
SampleThread threadObject1 = new SampleThread(); SampleThread threadObject2 = new SampleThread();
threadObject1.setName("first"); threadObject2.setName("second"); threadObject1.setPriority(4);
threadObject2.setPriority(Thread.MAX_PRIORITY); threadObject1.start();
threadObject2.start();
}
```

```
}
```

Output:

Inside SampleThread Inside SampleThread CurrentThread: second CurrentThread: first

Example2

```
class MultiThread 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[])
{
MultiThread m1=new MultiThread (); MultiThread m2=new MultiThread ();
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

SYNCHRONIZING THREADS SYNCHRONIZATION

- The java programming language supports multithreading.
- The problem of shared resources occurs when two or more threads get execute at the same time.
- In such a situation, we need some way to ensure that the shared resource will be accessed by only one thread at a time, and this is performed by using the concept called synchronization.
- The synchronization is the process of allowing only one thread to access a shared resource at a time.

UNDERSTANDING THE PROBLEM WITHOUT SYNCHRONIZATION

In this example, there is no synchronization, so output is inconsistent.

Example:

```
class Table
{ void printTable(int n)
{
//method not synchronized 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 TestSynchronization
{
```

```
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

100

10

200

15

300

20

400

25

500

Thread Synchronization

In java, the synchronization is achieved using the following concepts.

1. Mutual Exclusive

1. Synchronized method.

2. Synchronized block.

2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by two ways in java:

1. by synchronized method
2. by synchronized block

Java synchronized method

- If you declare any method as synchronized, it is known as synchronized method.
- When a method created using a synchronized keyword, it allows only one object to access it at a time.
- When an object calls a synchronized method, it put a lock on that method so that other objects or thread that are trying to call the same method must wait, until the lock is released.
- Once the lock is released on the shared resource, one of the threads among the waiting threads will be allocated to the shared resource.
- In the above image, initially the thread-1 is accessing the synchronized method and other threads (thread-2, thread-3, and thread-4) are waiting for the resource (synchronized method).
- When thread-1 completes it task, then one of the threads that are waiting is allocated with the synchronized method, in the above it is thread-3.

Example:

```
class Table

{

synchronized void printTable(int n)

{

for(int i = 1; i <= 10; i++) System.out.println(n + " * " + i + " = " + i*n);

}

}

class MyThread1 extends Thread
```

```
{
Table table = new Table(); int number;
MyThread1(Table table, int number)
{
this.table = table; this.number = number;
}
public void run()
{
table.printTable(number);
}
}
class MyThread2 extends Thread
{
Table table = new Table(); int number;
MyThread2(Table table, int number)
{
this.table = table; this.number = number;
}
public void run()
{
```

```
table.printTable(number);
}
}
class ThreadSynchronizationExample
{
public static void main(String[] args)
{
Table table = new Table();
MyThread1 thread1 = new MyThread1(table, 5); MyThread2 thread2 = new MyThread2(table, 10);
thread1.start();
thread2.start();
}
}
```

Output:

```
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
```

5 * 10 = 50

10 * 1 = 10

10 * 2 = 20

10 * 3 = 30

10 * 4 = 40

10 * 5 = 50

10 * 6 = 60

10 * 7 = 70

10 * 8 = 80

10 * 9 = 90

10 * 10 = 100

Synchronized Block in Java

- The synchronized block is used when we want to synchronize only a specific sequence of lines in a method.
- For example, let's consider a method with 20 lines of code where we want to synchronize only a sequence of 5 lines code, we use the synchronized block.
- If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.
- Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

synchronized (object reference expression) {

//code block }

Example of synchronized block

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 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

INTERTHREAD COMMUNICATION

- Inter thread communication is the concept where two or more threads communicate to solve the problem of polling.
- In java, polling is the situation to check some condition repeatedly, to take appropriate action, once the condition is true.
- That means, in inter-thread communication, a thread waits until a condition becomes true such that other threads can execute its task.
- The inter-thread communication allows the synchronized threads to communicate with each other.

Java provides the following methods to achieve inter thread communication.

- void wait( ) It makes the current thread to pause its execution until other thread in the same monitor calls notify( )
- void notify( ) It wakes up the thread that called wait( ) on the same object.
    - void notifyAll() It wakes up all the threads that called wait( ) on the same object. Let's look

at an example problem of producer and consumer.

- The producer produces the item and the consumer consumes the same.
- But here, the consumer cannot consume until the producer produces the item, and producer cannot produce until the consumer consumes the item that already been produced.
- So here, the consumer has to wait until the producer produces the item, and the producer also needs to wait until the consumer consumes the same.
- Here we use the inter-thread communication to implement the producer and consumer problem.

Example

```
class ItemQueue

{

int item;

boolean valueSet = false; synchronized int getItem()

{

while (!valueSet) try

{

wait();

}

catch (InterruptedException e)

{

System.out.println("InterruptedException  caught");
```

```
}
System.out.println("Consumed:" + item); valueSet = false; try
{
Thread.sleep(1000);
}
catch (InterruptedException e)
{
System.out.println("InterruptedException  caught");
}
notify(); return item;
}
synchronized void putItem(int item)
{
while (valueSet) try
{
wait();
}
catch (InterruptedException e)
{
System.out.println("InterruptedException  caught");
}
```

```
this.item = item; valueSet = true; System.out.println("Produced: " + item); try
{
Thread.sleep(1000);
}
catch (InterruptedException e)
{
System.out.println("InterruptedException  caught");
}
notify();
}
}
class Producer implements Runnable
{
ItemQueue itemQueue; Producer(ItemQueue itemQueue)
{
this.itemQueue = itemQueue;
new Thread(this, "Producer").start();
}
public void run()
{
int i = 0;
```

```
while(true)
{
itemQueue.putItem(i++);
}
}
}
class Consumer implements Runnable
{
ItemQueue itemQueue; Consumer(ItemQueue itemQueue)
{
this.itemQueue = itemQueue;
new Thread(this, "Consumer").start();
}
public void run()
{
while(true)
{
itemQueue.getItem();
}
}
}
class ProducerConsumer
{
```

```
public static void main(String args[])
{
ItemQueue itemQueue = new ItemQueue(); new Producer(itemQueue);
new Consumer(itemQueue);
}
}
```

THREADGROUP IN JAVA

- Java provides a convenient way to group multiple threads in a single object. In such a way, we can suspend, resume or interrupt a group of threads by a single method call.
- Java thread group is implemented by java.lang.ThreadGroup class.
- A Thread Group represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.
- A thread is allowed to access information about its own thread group, but it cannot access the information about its thread group's parent thread group or any other thread groups.

ThreadGroup Example

```
public class ThreadGroupDemo implements Runnable
{
                                                    public void run()
{
System.out.println(Thread.currentThread().getName());
}
public static void main(String[] args)
{
ThreadGroupDemo r = new ThreadGroupDemo(); ThreadGroup tg1 = new ThreadGroup("Parent
ThreadGroup");
```

```
Thread t1 = new Thread(tg1, r,"one"); t1.start();

Thread t2 = new Thread(tg1, r,"two"); t2.start();

Thread t3 = new Thread(tg1, r,"three"); t3.start();

System.out.println("Thread Group Name: "+tg1.getName());

}

}
```

Output :

```
one two three

Thread Group Name: Parent ThreadGroup
```

DAEMON THREAD IN JAVA

- In Java, daemon threads are low-priority threads that run in the background to perform tasks such as garbage collection or provide services to user threads.
- The life of a daemon thread depends on the mercy of user threads, meaning that when all user threads finish their execution, the Java Virtual Machine (JVM) automatically terminates the daemon thread.
- To put it simply, daemon threads serve user threads by handling background tasks and have no role other than supporting the main execution. Methods for Java Daemon thread by Thread class The java.lang.Thread class provides two methods for java daemon thread.

1 public void setDaemon(boolean status) is used to mark the current thread as daemon thread or user thread.

2 public boolean isDaemon() is used to check that current is daemon.

Example

```
public class TestDaemonThread1 extends Thread
```

```
{
public void run()
{
if(Thread.currentThread().isDaemon())
{
System.out.println("daemon thread work");
}
else
{
System.out.println("user thread work");
}
}
public static void main(String[] args)
{
TestDaemonThread1 t1=new TestDaemonThread1(); //creating thread TestDaemonThread1 t2=new
TestDaemonThread1(); TestDaemonThread1 t3=new TestDaemonThread1(); t1.setDaemon(true); //now t1
is daemon thread
t1.start(); //starting threads t2.start();
t3.start();
} }
```


Java Database Connectivity:

JDBC (Java Database Connectivity) is an API in Java that enables applications to interact with databases. It allows a Java program to connect to a database, execute queries, and retrieve and manipulate data. By providing a standard interface, JDBC ensures that Java applications can work with different relational databases like MySQL, Oracle, PostgreSQL, and more.

Types of Drivers:

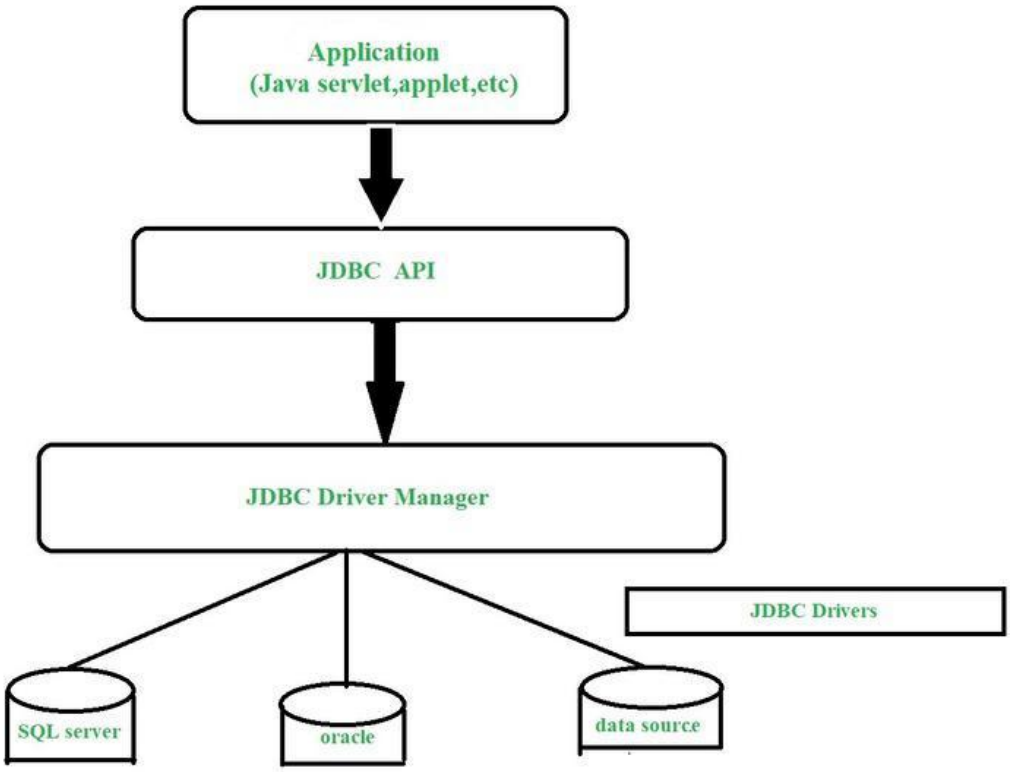
There are 4 types of JDBC drivers. It is part of the Java Standard Edition platform, from Oracle Corporation. It acts as a middle-layer interface between Java applications and databases.

The JDBC classes are contained in the Java Package java.sql and javax.sql.

JDBC helps you to write Java applications that manage these three programming activities:

1. Connect to a data source, like a database.
2. Send queries and update statements to the database
3. Retrieve and process the results received from the database in answer to your query

Structure of JDBC Driver



The above JDBC Driver structure illustrates the architecture of JDBC driver, where an application interacts with the JDBC API. The API communicates with the JDBC Driver Manager, which manages different database drivers e.g. SQL server, Oracle to establish database connectivity.

JDBC Drivers

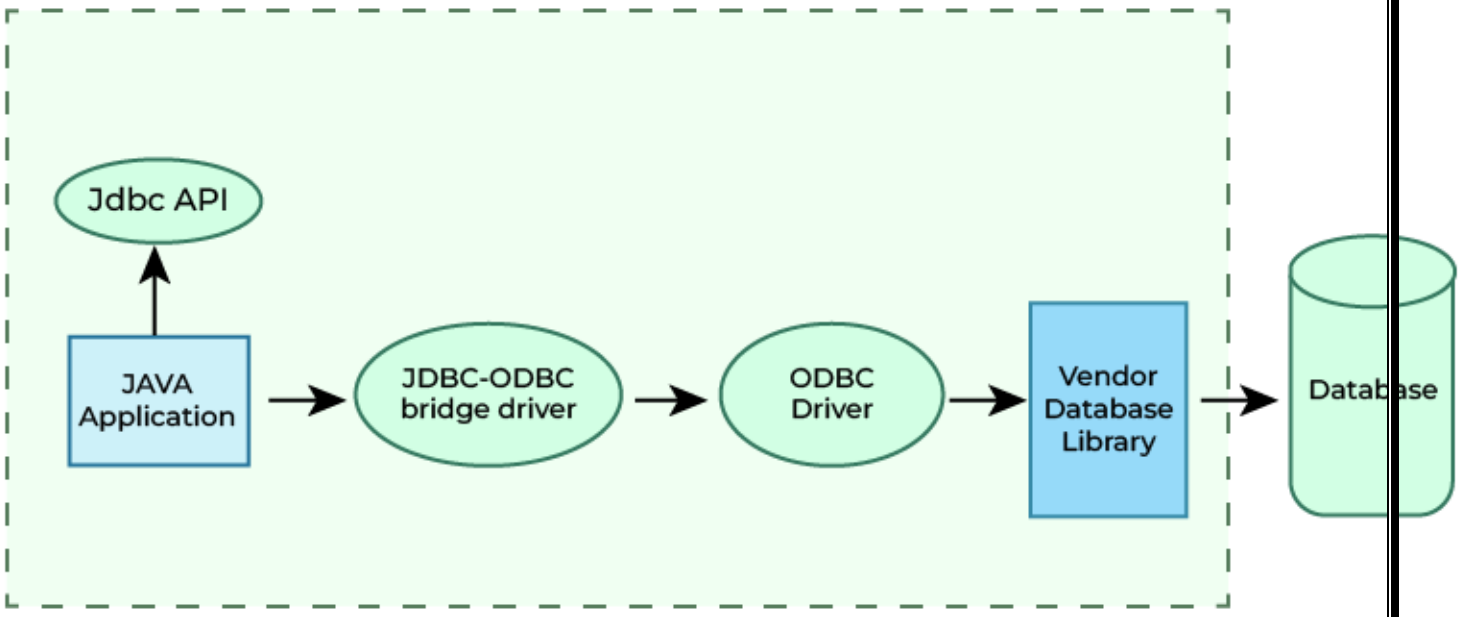
JDBC drivers are client-side adapters (installed on the client machine rather than the server) that translate requests from Java programs into a protocol understood by the DBMS. These drivers are software components that implement the interfaces in the JDBC API, allowing Java applications to interact with a database. Sun Microsystems (now Oracle) defines four types of JDBC drivers, which are outlined below:

1. Type-1 driver or JDBC-ODBC bridge driver

2. Type-2 driver or Native-API driver
3. Type-3 driver or Network Protocol driver
4. Type-4 driver or Thin driver

JDBC-ODBC Bridge Driver – Type 1 Driver

Type-1 driver or JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. Type-1 driver is also called Universal driver because it can be used to connect to any of the databases.



Advantages

- This driver software is built-in with JDK so no need to install separately.
- It is a database independent driver.

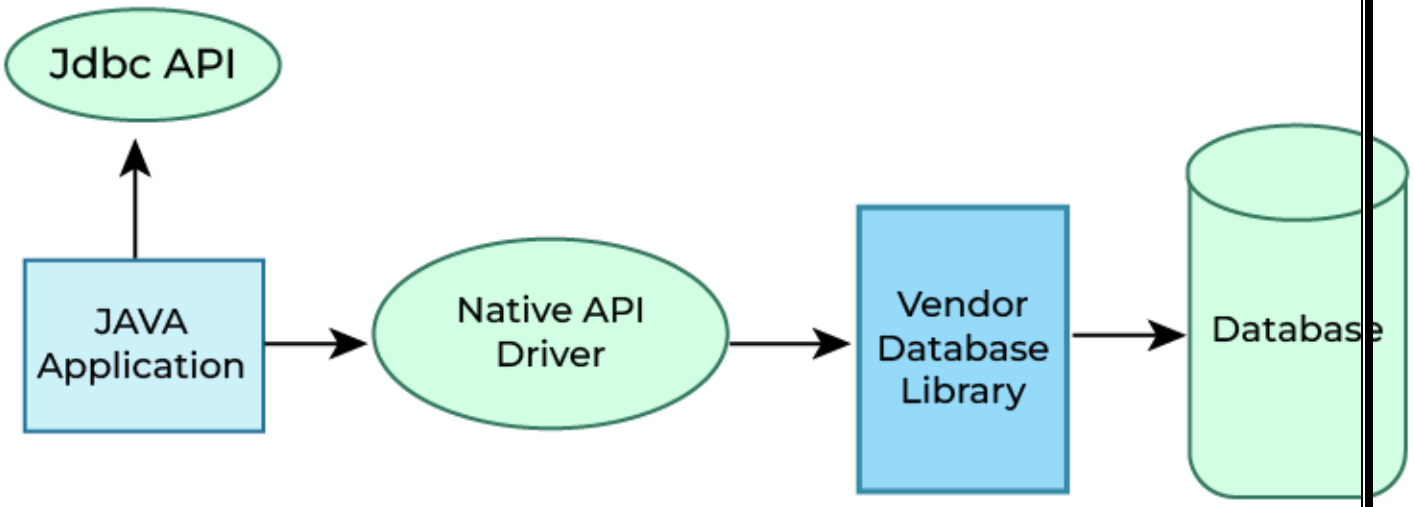
Disadvantages

- As a common driver is used in order to interact with different databases, the data transferred through this driver is not so secured.
- The ODBC bridge driver is needed to be installed in individual client machines.
- Type-1 driver isn't written in java, that's why it isn't a portable driver.

2. Native-API Driver – Type 2 Driver (Partially Java Driver)

The Native API driver uses the client -side libraries of the database. This driver converts JDBC method calls into native calls of the database API. In order to interact with different database, this driver needs their local API, that's why data transfer is much more secure as compared to type-1 driver. This driver is not

fully written in Java that is why it is also called Pure Java driver.



Advantage

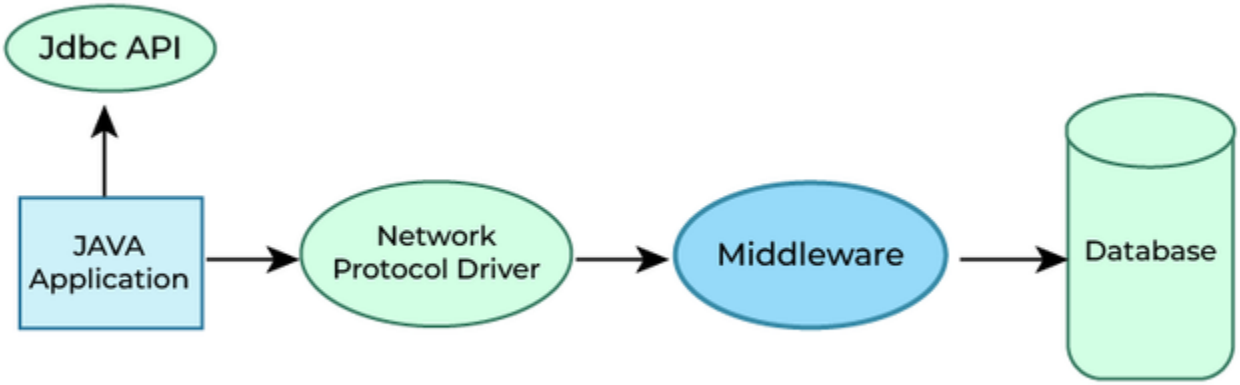
- Native-API driver gives better performance than JDBC-ODBC bridge driver.
- More secure compared to the type-1 driver.

Disadvantages

- Driver needs to be installed separately in individual client machines
- The Vendor client library needs to be installed on client machine.
- Type-2 driver isn't written in java, that's why it isn't a portable driver
- It is a database dependent driver.

3. Network Protocol Driver – Type 3 Driver (Fully Java Driver)

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. Here all the database connectivity drivers are present in a single server, hence no need of individual client-side installation.



Advantages

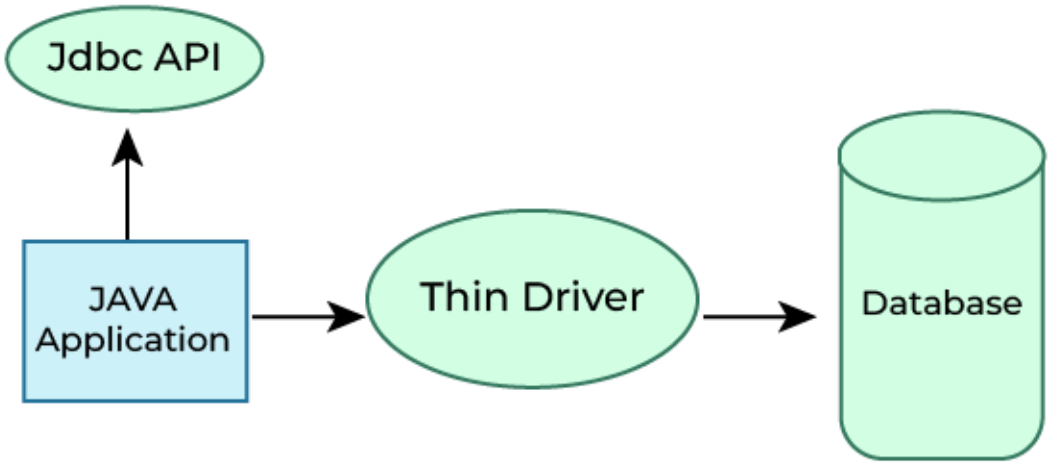
- Type-3 drivers are fully written in Java, hence they are portable drivers.
- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.
- Switch facility to switch over from one database to another database.

Disadvantages

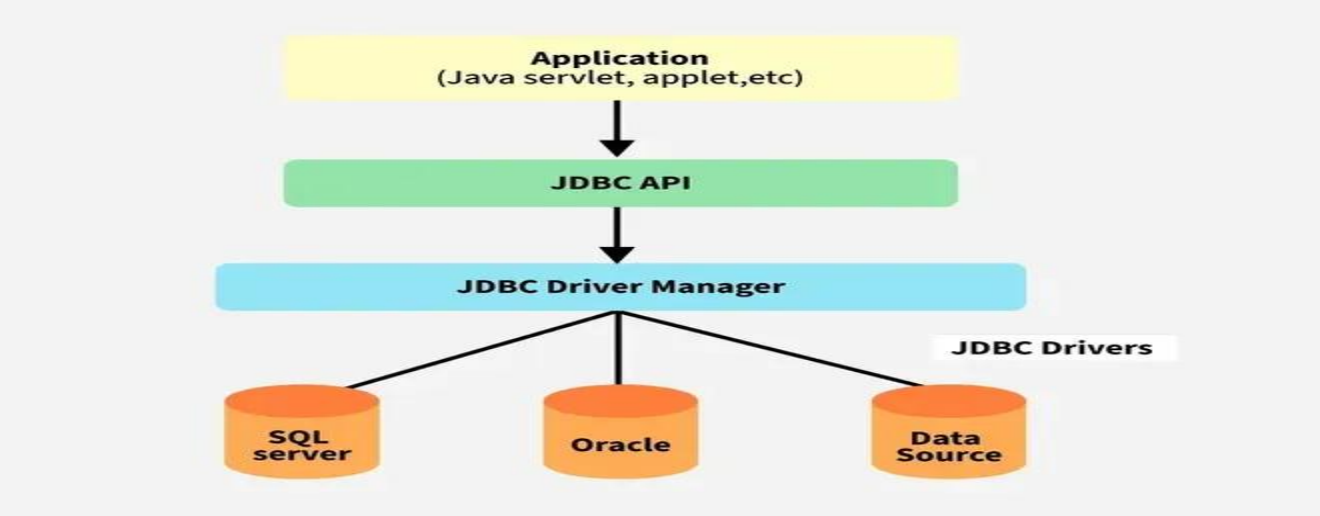
- Network support is required on client machine.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

4. Thin Driver – Type 4 Driver (Fully Java Driver)

Type-4 driver is also called native protocol driver. This driver interact directly with database. It does not require any native database library, that is why it is also known as Thin Driver.



JDBC architecture



Explanation:

Application: It is a Java applet or a servlet that communicates with a data source.

The JDBC API: It allows Java programs to execute SQL queries and retrieve results. Key interfaces include Driver, ResultSet, RowSet, PreparedStatement, and Connection. Important classes include DriverManager, Types, Blob, and Clob.

DriverManager: It plays an important role in the JDBC architecture. It uses some database-specific drivers to effectively connect enterprise applications to databases.

JDBC drivers: These drivers handle interactions between the application and the database.

The JDBC architecture consists of two-tier and three-tier processing models to access a database. They are as described below:

1. Two-Tier Architecture

A Java Application communicates directly with the database using a JDBC driver. Queries are sent to the database, and results are returned directly to the application. In a client/server setup, the user's machine (client) communicates with a remote database server.

Structure:

Client Application (Java) -> JDBC Driver -> Database

2. Three-Tier Architecture

In this, user queries are sent to a middle-tier services, which interacts with the database. The database results are processed by the middle tier and then sent back to the user.

Structure:

Client Application -> Application Server -> JDBC Driver -> Database

JDBC Classes and Interfaces:

Class/interface	Description
DriverManager	This class manages the JDBC drivers. You need to register your drivers to this. It provides methods such as registerDriver() and getConnection().
Driver	This interface is the Base interface for every driver class i.e. If you want to create a JDBC Driver of your own you need to implement this interface. If you load a Driver class (implementation of this interface), it will create an instance of itself and register with the driver manager.
Statement	This interface represents a static SQL statement.

Class/interface	Description
	Using the Statement object and its methods, you can execute an SQL statement and get the results of it. It provides methods such as execute(), executeBatch(), executeUpdate() etc. To execute the statements.
PreparedStatement	This represents a precompiled SQL statement. An SQL statement is compiled and stored in a prepared statement and you can later execute this multiple times. You can get an object of this interface using the method of the Connection interface named prepareStatement(). This provides methods such as executeQuery(), executeUpdate(), and execute() to execute the prepared statements and getXXX(), setXXX() (where XXX is the datatypes such as long int float etc..) methods to set and get the values of the bind variables of the prepared statement.
CallableStatement	Using an object of this interface you can execute the stored procedures. This returns single or multiple results. It will accept input parameters too. You can create a CallableStatement using the prepareCall() method of the Connection interface. Just like Prepared statement, this will also provide setXXX() and getXXX() methods to pass the input parameters and to get the output parameters of the procedures.
Connection	This interface represents the connection with a specific database. SQL statements are executed in the context of a connection. This interface provides methods such as close(), commit(), rollback(), createStatement(), prepareCall(), prepareStatement(), setAutoCommit() setSavepoint() etc.
ResultSet	This interface represents the database result set, a table which is generated by executing statements. This interface provides getter and update methods to retrieve and update its contents respectively.
ResultSetMetaData	This interface is used to get the information about the result set such as, number of columns, name of the column, data type of the column, schema of the result set, table name, etc

Class/interface	Description
	It provides methods such as getColumnCount(), getColumnName(), getColumnType(), getTableName(), getSchemaName() etc.

Basic steps in Developing JDBC Application:

Standard Steps followed for developing JDBC(JDBC4.X) Application

1. Load and register the Driver
2. Establish the Connection b/w java application and database
3. Create a Statement Object
4. Send and execute the Query
5. Process the result from ResultSet
6. Close the Connection

Step1: Load and register the Driver

A third-party DB vendor class that implements java.sql.Driver(I) is called a "Driver". This class Object we need to create and register it with JRE to set up the JDBC environment to run JDBC applications.

```
public class com.mysql.cj.jdbc.Driver extends
com.mysql.cj.jdbc.NonRegisteringDriver implements java.sql.Driver {

    public com.mysql.cj.jdbc.Driver() throws java.sql.SQLException;

    static {};

}
```

In MySQL Jar, the Driver class is implementing java.sql.Driver, so Driver class Object should be created and it should be registered to set up the JDBC environment inside JRE.

Step 2: Establish the Connection b/w java application and database

- public static Connection getConnection(String url, String username,String password) throws SQLException;
- public static Connection getConnection(String url, Properties) throws SQLException;
- public static Connection getConnection(String url) throws SQLException;

The below creates the Object of Connection interface.

```
Connection connection = DriverManager.getConnection(url,username,password);
```

getConnection(url,username,password) created an object of a class which implements Connection(I) that class object is collected by Connection(I). This feature in java refers to

- Abstraction(hiding internal services)

- Polymorphism(making code run in 1:M forms)

1. Can we create an Object for the Interface?

Answer: no

2. Can we create an Object for a class that implements an interface?

Answer: yes

Step 3: Create a Statement Object

- public abstract Statement createStatement() throws SQLException;
- public abstract Statement createStatement(int,int) throws SQLException;
- public abstract Statement createStatement(int,int,int) throws SQLException;

```
// create statement object

Statement statement = connection.createStatement();
```

Step 4: Send and execute the Query

From the DB administrator's perspective queries are classified into 5 types

1. DDL (Create table, alter table, drop table,..)
2. DML(Insert, update, delete)
3. DQL(select)
4. DCL(alter password,grant access)
5. TCL(commit,rollback,savepoint)

Read in detail here: DDL, DQL, DML, DCL and TCL Commands

According to the java developer perspective, we categorize queries into 2 types

- Select Query
- NonSelect Query

Methods for executing the Query are

- executeQuery() => for the select query we use this method.
- executeUpdate() => for insert, update and delete queries we use this method.
- execute() => for both select and non-select queries we use this method

```
public abstract ResultSet executeQuery(String sqlSelectQuery) throws
SQLException;

String sqlSelectQuery ="select sid,sname,sage,saddr from Student";

ResultSet resultSet = statement.executeQuery(sqlSelectQuery);
```

Step 5: Process the result from ResultSet

public abstract boolean next() throws java.sql.SQLException; => To check whether the next Record is available or not returns true if available otherwise returns false.

```
System.out.println("SID\tSNAME\tSAGE\tSADDR");
```

```
while(resultSet.next()){
    Integer id = resultSet.getInt(1);
    String name = resultSet.getString(2);
    Integer age = resultSet.getInt(3);
    String team = resultSet.getString(4);
    System.out.println(id+"\t"+name+"\t"+age+"\t"+team);
}
```

Step 6: Close the Connection

```
// Close the Connection
connection.close();
```

Creating a New Database and Table with JDBC:

The following steps are required to create a new Database using JDBC application −

- Import the packages − Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.** will suffice.
- Open a connection − Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with a database server.
- Execute a query − Requires using an object of type Statement for building and submitting an SQL statement to create a table in a seleted database.
- Clean up the environment − try with resources automatically closes the resources.

UNIT - V

GUI Programming with Swing : Introduction, limitations of AWT, MVC architecture, components, containers, Layout Manager Classes, Simple Applications using AWT and Swing.

Event Handling: The Delegation event model- Events, Event sources, Event Listeners, Event classes, Handling mouse and keyboard events, Adapter classes.

INTRODUCTION

Computer users today expect to interact with their computers using a graphical user interface (GUI). Java can be used to write GUI programs ranging from simple applets which run on a Web page to sophisticated stand- alone applications.

GUI programs differ from traditional “straight-through” programs that you have encountered in the first few chapters of this book. One big difference is that GUI programs are event-driven. That is, user actions such as clicking on a button or pressing a key on the keyboard generate events, and the program must respond to these events as they occur.

And of course, objects are everywhere in GUI programming. Events are objects. Colors and fonts are objects. GUI components such as buttons and menus are objects. Events are handled by instance methods contained in objects. In Java, GUI programming is object-oriented programming.

The Basic GUI Application

There are two basic types of GUI program in Java: stand-alone applications and applets. An applet is a program that runs in a rectangular area on a Web page. very complex. We will look at applets in the next

A stand-alone application is a program that runs on its own, without depending on a Web browser. You've been writing stand-alone applications all along. Any class that has a main() routine defines a stand-alone application; running the program just means executing this main() routine.

A GUI program offers a much richer type of user interface, where the user uses a mouse and keyboard to interact with GUI components such as windows, menus, buttons, check boxes, text input boxes, scroll bars, and so on. The main routine of a GUI program creates one or more such components and displays them on the computer screen. Very often, that's all it does. Once a GUI component has been created, it follows its own programming—programming that tells it how to draw itself on the screen and how to respond to events such as being clicked on by the user.

A GUI program doesn't have to be immensely complex. We can, for example, write a very simple GUI "Hello World" program that says "Hello" to the user, but does it by opening a window where the the greeting is displayed:

```
import javax.swing.JOptionPane;
public class HelloWorldGUI1
{
        public static void main(String[] args) {
        JOptionPane.showMessageDialog( null, "Hello World!" );
}

}
```

When this program is run, a window appears on the screen that contains the message "Hello World!". The window also contains an "OK" button for the user to click after reading the message. When the user clicks this button, the window closes and the program ends. By the way, this program can be placed in a file named HelloWorldGUI1.java, compiled, and run just like any other Java program.

SWING INTRODUCTION

- Java 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, JColorChooser etc.

LIMITATIONS OF AWT

- The buttons of AWT does not support pictures.
- It is heavyweight in nature.

- Two very important components trees and tables are not present.
- Extensibility is not possible as it is platform dependent

MVC ARCHITECTURE

In real time applications, in the case of server side programming one must follow the architecture to develop a distributed application.To develop any distributed application, it is always recommended to follow either 3-tier architecture or 2-tier architecture or n-tier architecture.

3-tier architecture is also known as MVC architecture.

M stands for Model (database programming),
V stands for View (client side programming, HTML/AWT/APPLET/Swing/JSP)
C stands for Controller (server side programming, Servlets).

Model :

- This is the data layer which consists of the business logic of the system.
- It consists of all the data of the application
- It also represents the state of the application.
- It consists of classes which have the connection to the database.
- The controller connects with model and fetches the data and sends to the view layer.
- The model connects with the database as well and stores the data into a database which is connected to it.

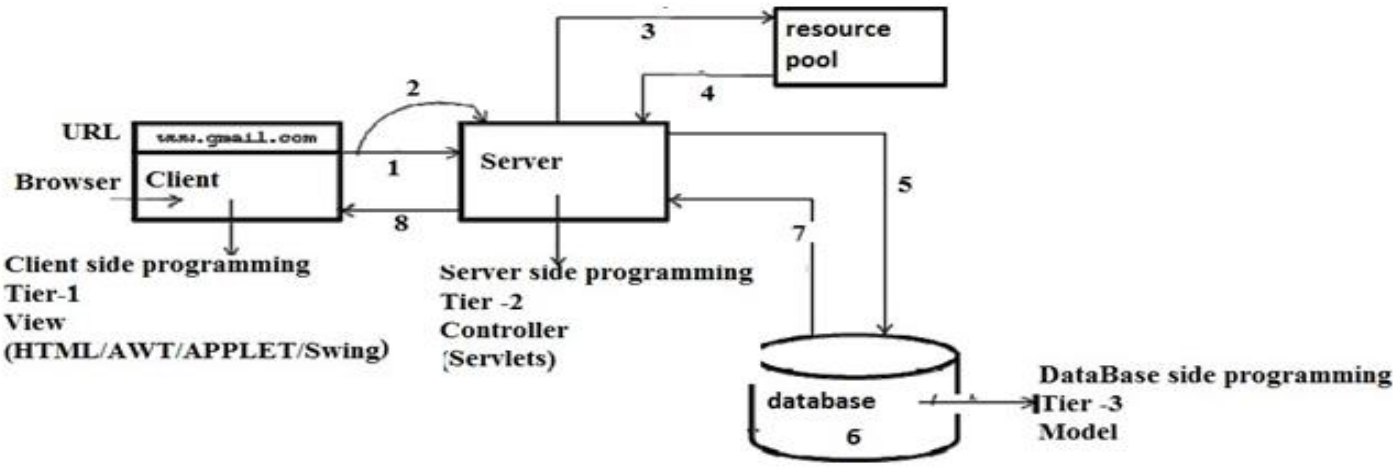
View :

- This is a presentation layer.
- It consists of HTML, JSP, etc. into it.
- It normally presents the UI of the application.
- It is used to display the data which is fetched from the controller which in turn fetching data from model layer classes.
- This view layer shows the data on UI of the application.

Controller:

- It acts as an interface between View and Model.
- It intercepts all the requests which are coming from the view layer.
- It receives the requests from the view layer and processes the requests and does the necessary validation for the request.
- This requests is further sent to model layer for data processing, and once the request is processed, it sends back to the controller with required information and displayed accordingly by the view.

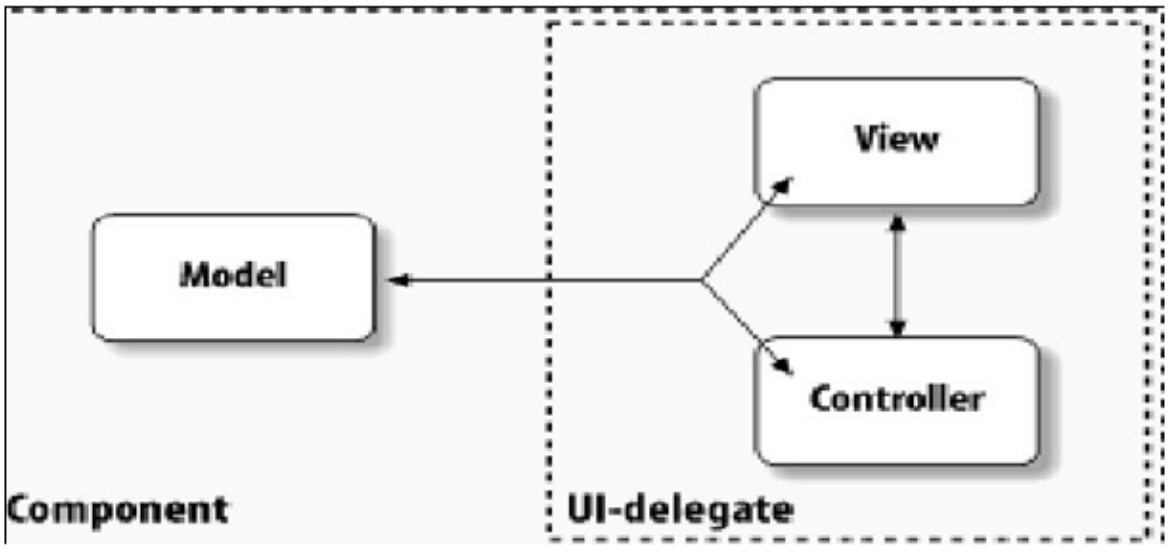
The general architecture of MVC or 3-tier:



1. Client makes a request.
2. Server side program receives the request.
3. The server looks for or search for the appropriate resource in the resource pool.
4. If the resource is not available server side program displays a user friendly message (page cannot be displayed). If the resource is available, that program will execute gives its result to server, server interns gives response to that client who makes a request.
5. When server want to deals with database to retrieve the data, server side program sends a request to the appropriate database.
6. Database server receives the server request and executes that request.
7. The database server sends the result back to server side program for further processing.
8. The server side program is always gives response to 'n' number of clients concurrently.

MVC in Swing

Swing actually uses a simplified variant of the MVC design called the model-delegate . This design combines the view and the controller object into a single element, the UI delegate , which draws the component to the screen and handles GUI events. Bundling graphics capabilities and event handling is somewhat easy in Java, since much of the event handling is taken care of in AWT. As you might expect, the communication between the model and the UI delegate then becomes a two-way street, as shown in Figure below.



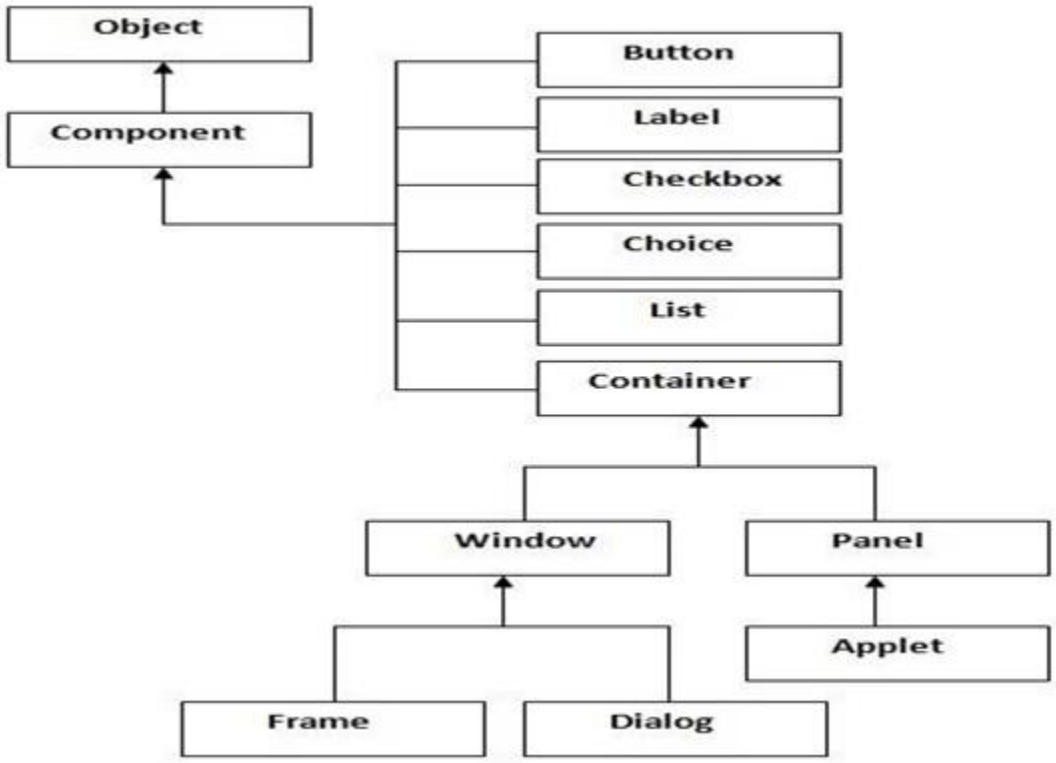
So let's review: each Swing component contains a model and a UI delegate. The model is responsible for maintaining information about the component's state. The UI delegate is responsible for maintaining information about how to draw the component on the screen. In addition, the UI delegate (in conjunction with AWT) reacts to various events that propagate through the component.

Note that the separation of the model and the UI delegate in the MVC design is extremely advantageous. One unique aspect of the MVC architecture is the ability to tie multiple views to a single model. For example, if you want to display the same data in a pie chart and in a table, you can base the views of two components on a single data model. That way, if the data needs to be changed, you can do so in only one place—the views update themselves accordingly

COMPONENTS

Component is an object having a graphical representation that can be displayed on the screen and that can interact with the user. For examples buttons, checkboxes, list and scrollbars of a graphical user interface.

A Component is an abstract super class for GUI controls and it represents an object with graphical representation.



Every AWT controls inherits properties from Component class

Component	Description

Label	The easiest control to use is a label. A label is an object of type Label, and it contains a string,which it displays. Labels are passive controls that do not support any interaction with theuser. Label defines the following constructors
Button	This class creates a labeled button.
Check Box	A check box is a graphical component that can be in either an on (true) or off (false) state.
Check Box Group	The CheckboxGroup class is used to group the set of checkbox.
List	The List component presents the user with a scrolling list of text items.
Text Field	A TextField object is a text component that allows for the editing of a single line of text.
Text Area	A TextArea object is a text component that allows for the editing of a multiple lines of text.
Choice	A Choice control is used to show pop up menu of choices. Selected choice is shown on the top of the menu.
Canvas	A Canvas control represents a rectangular area where application can draw something or can receive inputs created by user.
Image	An Image control is superclass for all image classes representing graphical images.
Scroll Bar	A Scrollbar control represents a scroll bar component in order to enable user to select from range of values.
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.
File Dialog	A FileDialog control represents a dialog window from which the user can select a file.

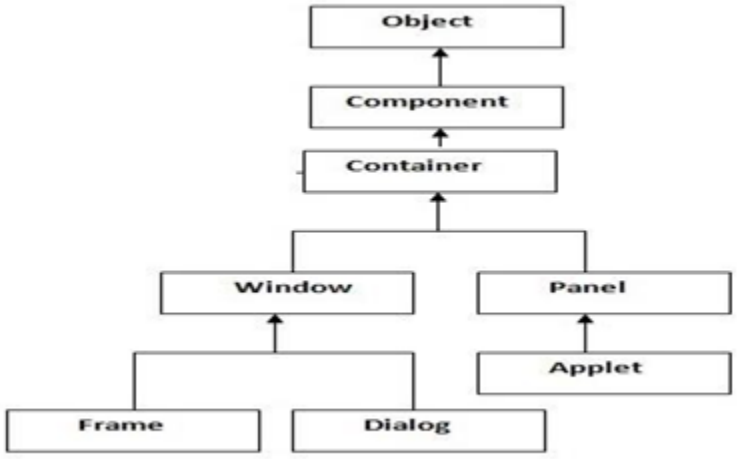
Commonly used Methods of Component class:

Method	Description
public void add(Component c)	inserts a component on this component.
public void setSize(intwidth,int height)	sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.
void remove(Component obj)	Here, obj is a reference to the control you want to remove.
void removeAll( ).	You can remove all controls by

CONTAINERS

Abstract Windowing Toolkit (AWT): Abstract Windowing Toolkit (AWT) is used for GUI programming in java.

AWT Container Hierarchy:



Container:
The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as container.

Window:
The window is the container that have no borders and menubars. You must use frame, dialog or another window for creating a window.

Panel:
The Panel is the container that doesn't contain title bar and MenuBars. It can have other components like button, textfield etc.

Frame:
The Frame is the container that contain title bar and can have MenuBars. It can have other components like button, textfield etc.

There are two ways to create a frame:

a. By extending Frame class (inheritance)
b. By creating the object of Frame class (association)

Example program to create a frame by extending Frame class (inheritance)

```
import java.awt.*;
class First extends Frame
{
First()
{
Button b=new Button("click me");
b.setBounds(30,100,80,30);/*setting button position public void setBounds(int xaxis, int yaxis, int width, int height); have been used in the above example that sets the position of the button.*/
add(b);//adding button into frame
setSize(300,300);//frame size 300 width and 300 height setLayout(null);//no layout now bydefault BorderLayout setVisible(true);//now frame willbe visible, bydefault not visible
}
public static void main(String args[])
```

```
{
First f=new First();
}
}
```

2.Example program to create a frame by creating the object of Frame class

```
import java.awt.*; class First2{ First2(){
Frame f=new Frame();
Button b=new Button("click me"); b.setBounds(30,50,80,30); f.add(b);
f.setSize(300,300); f.setLayout(null); f.setVisible(true);
}
public static void main(String args[]){ First2 f=new First2();
}
}
```

5.1 LAYOUT MANAGERS

The LayoutManagers are used to arrange components in a particular manner. The Java LayoutManagers facilitates us to control the positioning and size of the components in GUI forms. LayoutManager is an interface that is implemented by all the classes of layout managers. There are the following classes that represent the layout managers:

1. java.awt.BorderLayout
2. java.awt.FlowLayout
3. java.awt.GridLayout
4. java.awt.CardLayout
5. java.awt.GridBagLayout

5.6.1 BorderLayout

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

BorderLayout provides five constants for each region:

1. public static final int NORTH
2. public static final int SOUTH
3. public static final int EAST
4. public static final int WEST
5. public static final int CENTER

Constructors of BorderLayout class:

BorderLayout():

creates a border layout but with no gaps between the components.

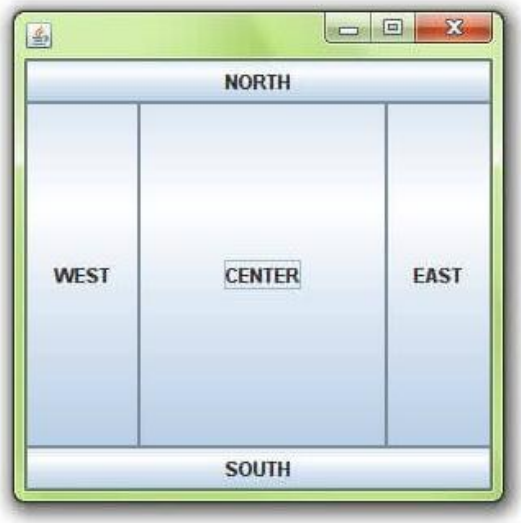
BorderLayout(int hgap, int vgap):

creates a border layout with the given horizontal and vertical gaps between the components.

Example of BorderLayout class: (FileName: Border.java)

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

public class Border
{
JFrame f;
Border()
{
   f = new JFrame();

    // creating buttons
   JButton b1 = new JButton("NORTH");; // the button will be labeled as NORTH
   JButton b2 = new JButton("SOUTH");; // the button will be labeled as SOUTH
   JButton b3 = new JButton("EAST");; // the button will be labeled as EAST
   JButton b4 = new JButton("WEST");; // the button will be labeled as WEST
   JButton b5 = new JButton("CENTER");; // the button will be labeled as CENTER

   f.add(b1, BorderLayout.NORTH); // b1 will be placed in the North Direction
   f.add(b2, BorderLayout.SOUTH);  // b2 will be placed in the South Direction
   f.add(b3, BorderLayout.EAST);  // b2 will be placed in the East Direction
   f.add(b4, BorderLayout.WEST);  // b2 will be placed in the West Direction
   f.add(b5, BorderLayout.CENTER);  // b2 will be placed in the Center

   f.setSize(300, 300);
   f.setVisible(true);
}
public static void main(String[] args) {
   new Border();
}
}
```

Output:



5.6.2 FlowLayout

The Java FlowLayout class is used to arrange the components in a line, one after another (in a flow). It is the default layout of the applet or panel.

Fields of FlowLayout class

public static final int LEFT
public static final int RIGHT
public static final int CENTER
public static final int LEADING
public static final int TRAILING

Constructors of FlowLayout class

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

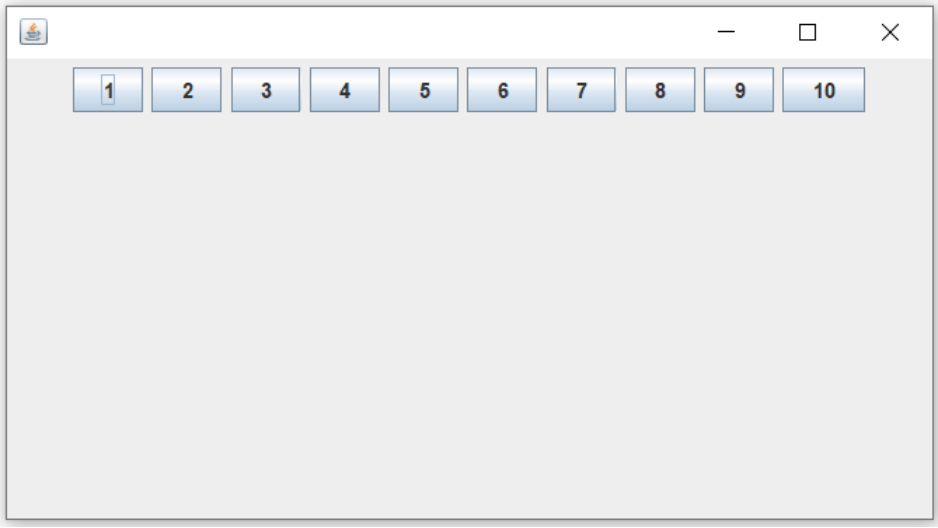
Example of FlowLayout class: Using FlowLayout() constructor

(FileName: FlowLayoutExample.java)

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

public class FlowLayoutExample
{

JFrame frameObj;

// constructor
FlowLayoutExample()
{
   // creating a frame object
```

```
    frameObj = new JFrame();

     // creating the buttons
    JButton b1 = new JButton("1");
    JButton b2 = new JButton("2");
    JButton b3 = new JButton("3");
    JButton b4 = new JButton("4");
    JButton b5 = new JButton("5");
    JButton b6 = new JButton("6");
    JButton b7 = new JButton("7");
    JButton b8 = new JButton("8");
    JButton b9 = new JButton("9");
    JButton b10 = new JButton("10");


    // adding the buttons to frame
    frameObj.add(b1); frameObj.add(b2); frameObj.add(b3); frameObj.add(b4);
    frameObj.add(b5); frameObj.add(b6);  frameObj.add(b7);  frameObj.add(b8);
    frameObj.add(b9);  frameObj.add(b10);

    // parameter less constructor is used
    // therefore, alignment is center
    // horizontal as well as the vertical gap is 5 units.
    frameObj.setLayout(new FlowLayout());

    frameObj.setSize(300, 300);
    frameObj.setVisible(true);
}

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

Output:



5.6.3 GridLayout

The Java GridLayout class is used to arrange the components in a rectangular grid. One component is displayed in each rectangle.

Constructors of GridLayout class

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

Example of GridLayout class: Using GridLayout() Constructor

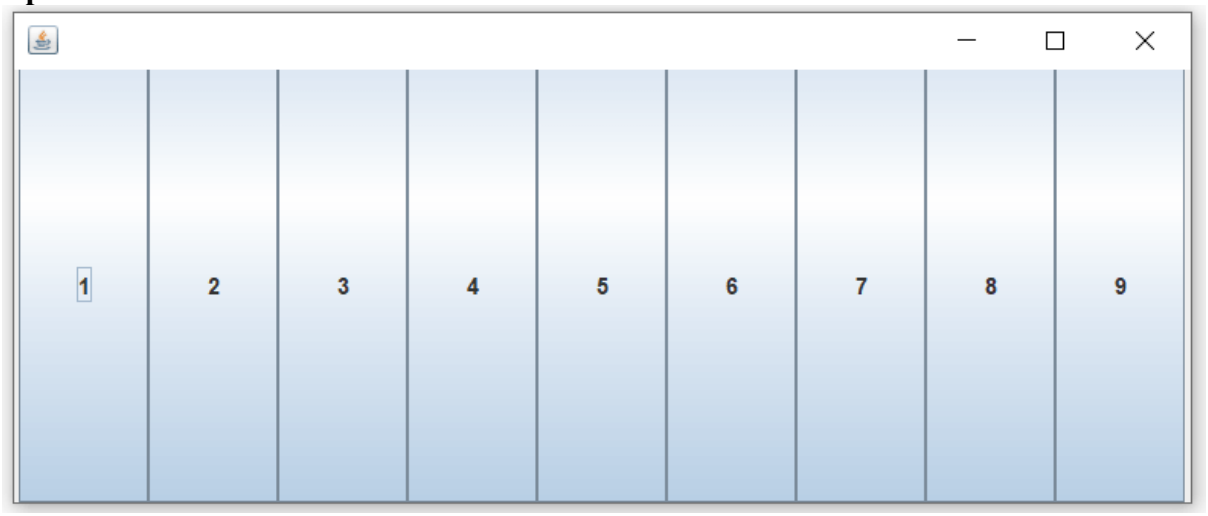
The GridLayout() constructor creates only one row. The following example shows the usage of the parameterless constructor.

FileName: GridLayoutExample.java

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

public class GridLayoutExample
{
JFrame frameObj;

// constructor
GridLayoutExample()
{
frameObj = new JFrame();

// creating 9 buttons
JButton btn1 = new JButton("1");
JButton btn2 = new JButton("2");
JButton btn3 = new JButton("3");
```

```
JButton btn4 = new JButton("4");
JButton btn5 = new JButton("5");
JButton btn6 = new JButton("6");
JButton btn7 = new JButton("7");
JButton btn8 = new JButton("8");
JButton btn9 = new JButton("9");

// adding buttons to the frame
// since, we are using the parameterless constructor, therfore;
// the number of columns is equal to the number of buttons we
// are adding to the frame. The row count remains one.
frameObj.add(btn1); frameObj.add(btn2); frameObj.add(btn3);
frameObj.add(btn4); frameObj.add(btn5); frameObj.add(btn6);
frameObj.add(btn7); frameObj.add(btn8); frameObj.add(btn9);

// setting the grid layout using the parameterless constructor
frameObj.setLayout(new GridLayout());


frameObj.setSize(300, 300);
frameObj.setVisible(true);
}

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

Output:

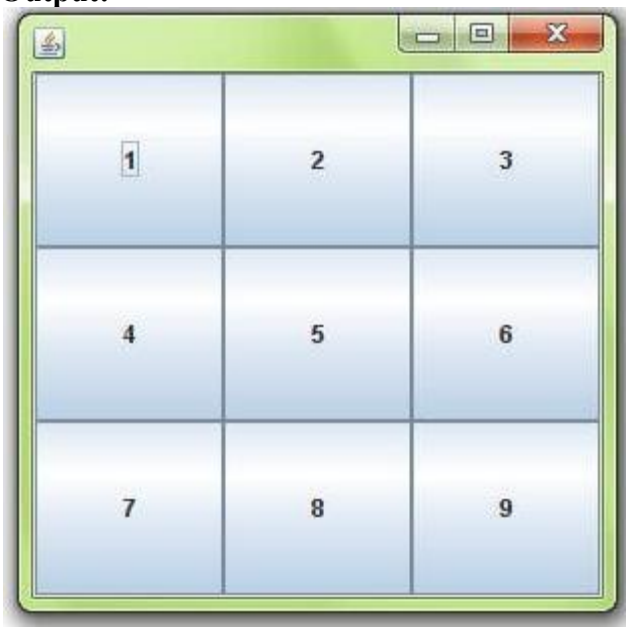


Example of GridLayout class: Using GridLayout(int rows, int columns) Constructor
FileName: MyGridLayout.java

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

```
public class MyGridLayout{
JFrame f;
MyGridLayout(){
   f=new JFrame();
   JButton b1=new JButton("1");
   JButton b2=new JButton("2");
   JButton b3=new JButton("3");
   JButton b4=new JButton("4");
   JButton b5=new JButton("5");
   JButton b6=new JButton("6");
   JButton b7=new JButton("7");
   JButton b8=new JButton("8");
   JButton b9=new JButton("9");
    // adding buttons to the frame
   f.add(b1); f.add(b2); f.add(b3);
   f.add(b4); f.add(b5); f.add(b6);
   f.add(b7); f.add(b8); f.add(b9);

   // setting grid layout of 3 rows and 3 columns
   f.setLayout(new GridLayout(3,3));
   f.setSize(300,300);
   f.setVisible(true);
}
public static void main(String[] args) {
   new MyGridLayout();
}
}
```

Output:



5.6.4 CardLayout

The Java CardLayout class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout.

Constructors of CardLayout Class
CardLayout(): creates a card layout with zero horizontal and vertical gap.
CardLayout(int hgap, int vgap): creates a card layout with the given horizontal and vertical gap.

Commonly Used Methods of CardLayout Class

public void next(Container parent): is used to flip to the next card of the given container.
public void previous(Container parent): is used to flip to the previous card of the given container.
public void first(Container parent): is used to flip to the first card of the given container.
public void last(Container parent): is used to flip to the last card of the given container.
public void show(Container parent, String name): is used to flip to the specified card with the given name.

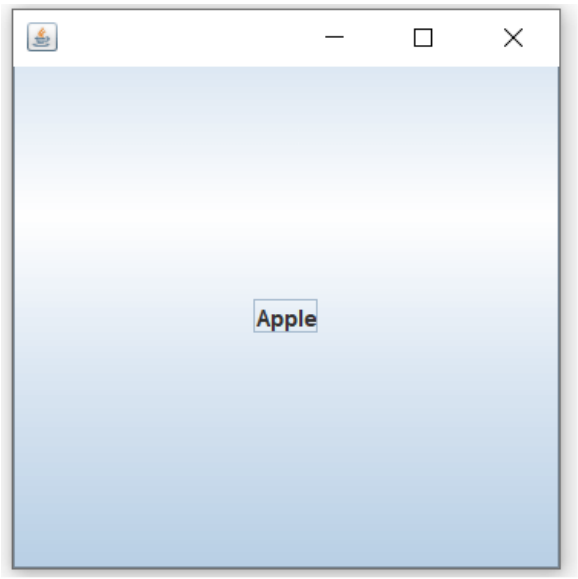
Example of CardLayout Class: Using Default Constructor
The following program uses the next() method to move to the next card of the container.

FileName: CardLayoutExample1.java

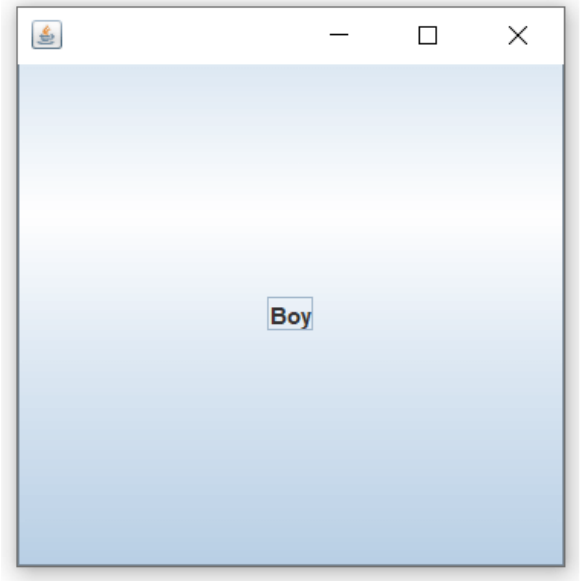
```
// import statements
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class CardLayoutExample1 extends JFrame implements ActionListener
{

CardLayout crd;

// button variables to hold the references of buttons
JButton btn1, btn2, btn3;
Container cPane;

// constructor of the class
CardLayoutExample1()
{

cPane = getContentPane();

//default constructor used
// therefore, components will
// cover the whole area
crd = new CardLayout();

cPane.setLayout(crd);

// creating the buttons
btn1 = new JButton("Apple");
btn2 = new JButton("Boy");
btn3 = new JButton("Cat");
```

```
// adding listeners to it
btn1.addActionListener(this);
btn2.addActionListener(this);
btn3.addActionListener(this);

cPane.add("a", btn1); // first card is the button btn1
cPane.add("b", btn2); // first card is the button btn2
cPane.add("c", btn3);  // first card is the button btn3

}
public void actionPerformed(ActionEvent e)
{
// Upon clicking the button, the next card of the container is shown
// after the last card, again, the first card of the container is shown upon clicking
crd.next(cPane);
}

// main method
public static void main(String argvs[])
{
// creating an object of the class CardLayoutExample1
CardLayoutExample1 crdl = new CardLayoutExample1();

// size is 300 * 300
crdl.setSize(300, 300);
crdl.setVisible(true);
crdl.setDefaultCloseOperation(EXIT_ON_CLOSE);
}
}
```

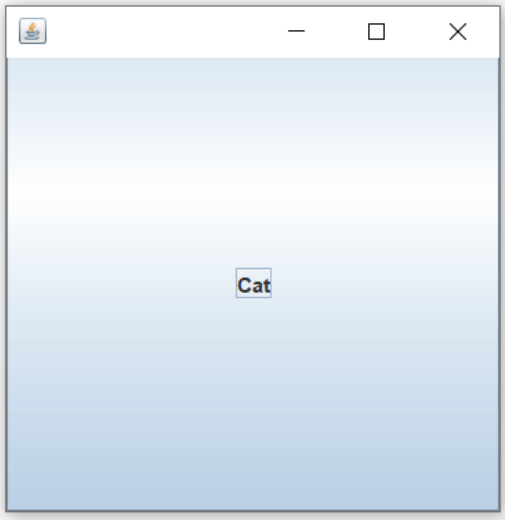
Output:



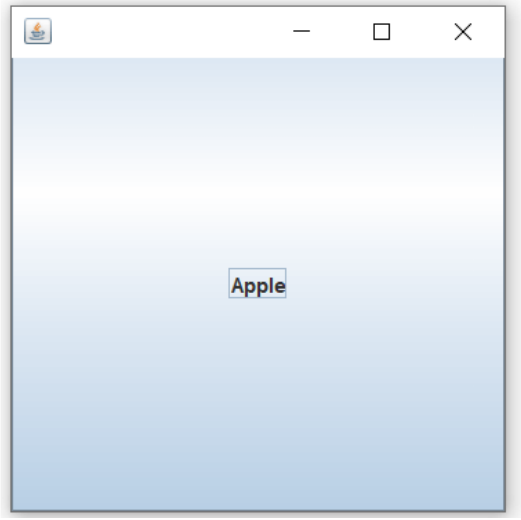
When the button named apple is clicked, we get



When the boy button is clicked, we get



Again, we reach the first card of the container if the cat button is clicked, and the cycle continues.



5.6.5 GridBagLayout

The Java GridBagLayout class is used to align components vertically, horizontally or along their baseline. The components may not be of the same size. Each GridBagLayout object maintains a dynamic, rectangular grid of cells. Each component occupies one or more cells known as its display area. Each component associates an instance of GridBagConstraints. With the help of the constraints object, we arrange the component's display area on the grid. The GridBagLayout manages each component's minimum and preferred sizes in order to determine the component's size. GridBagLayout components are also arranged in the rectangular grid but can have many different sizes and can occupy multiple rows or columns.

Constructor

GridBagLayout(): The parameterless constructor is used to create a grid bag layout manager.

GridBagLayout Fields

Modifier and Type	Field	Description
double[]	columnWeights	It is used to hold the overrides to the column weights.
int[]	columnWidths	It is used to hold the overrides to the column minimum width.
protected Hashtable<Component,GridBagConstraints>	comptable	It is used to maintains the association between a component and its gridbag constraints.
protected GridBagConstraints	defaultConstraints	It is used to hold a gridbag constraints instance containing the default values.
protected GridBagLayoutInfo	layoutInfo	It is used to hold the layout information for the gridbag.
protected static int	MAXGRIDSIZE	No longer in use just for backward compatibility
protected static int	MINSIZE	It is smallest grid that can be laid out by the grid bag layout.
protected static int	PREFERREDSIZE	It is preferred grid size that can be laid out by the grid bag layout.
int[]	rowHeights	It is used to hold the overrides to the row minimum heights.
double[]	rowWeights	It is used to hold the overrides to the row weights.

GridBagLayout Methods

Modifier and Type	Method	Description
void	addLayoutComponent(Component comp, Object constraints)	It adds specified component to the layout, using the specified constraints object.
void	addLayoutComponent(String name, Component comp)	It has no effect, since this layout manager does not use a per-component string.
protected void	adjustForGravity(GridBagConstraints constraints, Rectangle r)	It adjusts the x, y, width, and height fields to the correct values depending on the constraint geometry and pads.
protected void	AdjustForGravity(GridBagConstraints constraints, Rectangle r)	This method is for backwards compatibility only
protected void	arrangeGrid(Container parent)	Lays out the grid.
protected void	ArrangeGrid(Container parent)	This method is obsolete and supplied for backwards

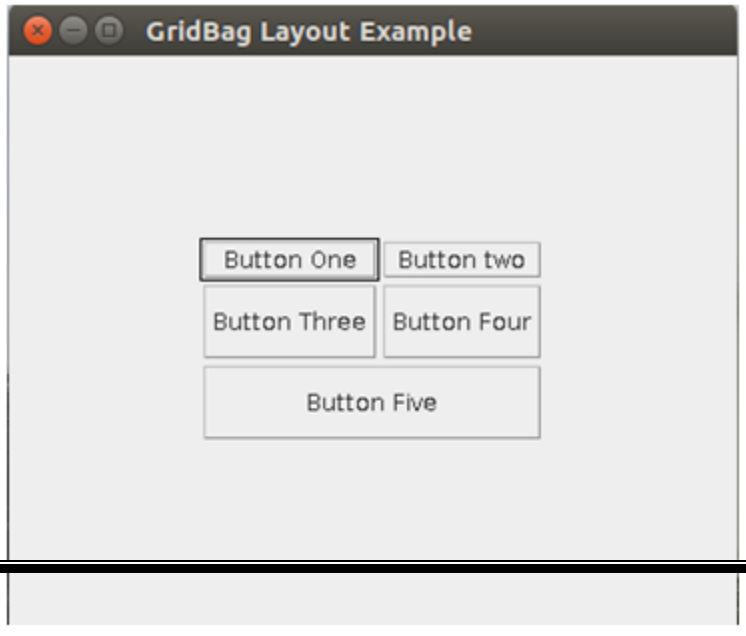
		compatibility
GridBagConstraints	getConstraints(Component comp)	It is for getting the constraints for the specified component.
float	getLayoutAlignmentX(Container parent)	It returns the alignment along the x axis.
float	getLayoutAlignmentY(Container parent)	It returns the alignment along the y axis.
int[][]	getLayoutDimensions()	It determines column widths and row heights for the layout grid.
protected GridBagLayoutInfo	getLayoutInfo(Container parent, int sizeflag)	This method is obsolete and supplied for backwards compatibility.
protected GridBagLayoutInfo	GetLayoutInfo(Container parent, int sizeflag)	This method is obsolete and supplied for backwards compatibility.
Point	getLayoutOrigin()	It determines the origin of the layout area, in the graphics coordinate space of the target container.
double[][]	getLayoutWeights()	It determines the weights of the layout grid's columns and rows.
protected Dimension	getMinSize(Container parent, GridBagLayoutInfo info)	It figures out the minimum size of the master based on the information from getLayoutInfo.
protected Dimension	GetMinSize(Container parent, GridBagLayoutInfo info)	This method is obsolete and supplied for backwards compatibility only

Example 1
FileName: GridBagLayoutExample.java

```
import java.awt.Button;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;

import javax.swing.*;
public class GridBagLayoutExample extends JFrame{
```

```
public static void main(String[] args) {
        GridBagLayoutExample a = new GridBagLayoutExample();
    }
    public GridBagLayoutExample() {
GridBagLayoutgrid = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
        setLayout(grid);
        setTitle("GridBag Layout Example");
        GridBagLayout layout = new GridBagLayout();
this.setLayout(layout);
gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.gridx = 0;
gbc.gridy = 0;
this.add(new Button("Button One"), gbc);
gbc.gridx = 1;
gbc.gridy = 0;
this.add(new Button("Button two"), gbc);
gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.ipady = 20;
gbc.gridx = 0;
gbc.gridy = 1;
this.add(new Button("Button Three"), gbc);
gbc.gridx = 1;
gbc.gridy = 1;
this.add(new Button("Button Four"), gbc);
gbc.gridx = 0;
gbc.gridy = 2;
gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.gridwidth = 2;
this.add(new Button("Button Five"), gbc);
        setSize(300, 300);
        setPreferredSize(getSize());
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);

    }

        }
```

Output:



EVENT HANDLING

- In general we 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.

EVENTS

- The Events are the objects that define state change in a source.
- An event can be generated as a reaction of a user while interacting with GUI elements.
- Some of the event generation activities are moving the mouse pointer, clicking on a button, pressing the keyboard key, selecting an item from the list, and so on.
- We can also consider many other user operations as events.

EVENT SOURCES

- A source is an object that causes and generates an event.
- It generates an event when the internal state of the object is changed.
- The sources are allowed to generate several different types of events.
- A source must register a listener to receive notifications for a specific event.
- Each event contains its registration method.

Syntax

public void addTypeListener (TypeListener e1)

From the above syntax, the Type is the name of the event, and e1 is a reference to the event listener.

- For example, for a keyboard event listener, the method will be called as addKeyListener().
- For the mouse event listener, the method will be called as addMouseMotionListener(). When an event is triggered using the respected source, all the events will be notified to registered listeners and receive the event object.
- This process is known as event multicasting.

EVENT LISTENERS

- 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.

EVENTS	SOURCE	LISTENERS
Action Event	Button, List, MenuItem, Text field	ActionListener
Component Event	Component	Component Listener
Focus Event	Component	FocusListener
Item Event	Checkbox, CheckboxMenuItem, Choice, List	ItemListener
Key Event	when input is received from keyboard	KeyListener
Text Event	Text Component	TextListener
Window Event	Window	WindowListener
Mouse Event	Mouse related event	MouseListener

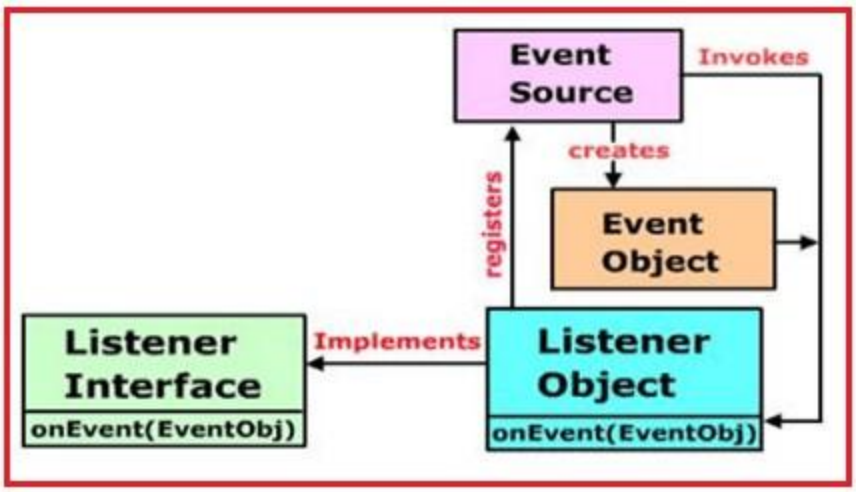
EVENT CLASSES AND LISTENER INTERFACES

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 or set visible	ComponentEventListener
ContainerEvent	generated when component is added or removed from container	ContainerListener
AdjustmentEvent	generated when scroll bar is manipulated	AdjustmentListener

DELEGATION EVENT MODEL IN JAVA

- The Delegation Event model is defined to handle events in GUI programming languages.
- The GUI stands for Graphical User Interface, where a user graphically/visually interacts with the system.

- The GUI programming is inherently event-driven; whenever a user initiates an activity such as a mouse activity, clicks, scrolling, etc., each is known as an event that is mapped to a code to respond to functionality to the user. This is known as event handling. The below image demonstrates the event processing.
- In this model, a source generates an event and forwards it to one or more listeners.
- The listener waits until it receives an event. Once it receives the event, it is processed by the listener and returns it.



REGISTRATION METHODS

For registering the component with the Listener, many classes provide the registration methods.

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
- 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 Mouse Event class and Mouse Listener interface.

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);

Example

```
import java.awt.*; import java.awt.event.*;
public class MouseListenerExample extends Frame implements MouseListener
{
Label l; MouseListenerExample()
{
 addMouseListener(this); l=new Label(); l.setBounds(20,50,100,20); add(l);
 setSize(300,300); setLayout(null); 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:

EVENT HANDLING FOR KEYBOARD

The Java KeyListener is notified whenever you change the state of key. It is notified against KeyEvent.

The KeyListener interface is found in java.awt.event package. It has three methods.

Methods of KeyListener interface

1. public abstract void keyPressed(KeyEvent e);
2. public abstract void keyReleased(KeyEvent e);
3. public abstract void keyTyped(KeyEvent e);

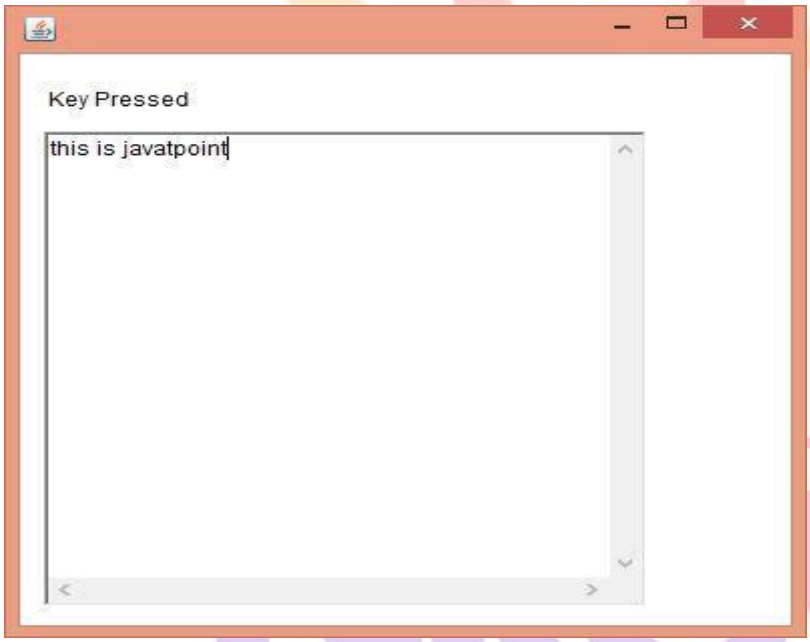
EXAMPLE

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

public class KeyListenerExample extends Frame implements KeyListener

{

Label l;
```

```
TextArea area; KeyListenerExample()
{
l=new Label(); l.setBounds(20,50,100,20); area=new TextArea(); area.setBounds(20,80,300, 300);
area.addKeyListener(this); add(l);
add(area); setSize(400,400); setLayout(null); setVisible(true);
}
public void keyPressed(KeyEvent e)
{
l.setText("Key Pressed");
}
public void keyReleased(KeyEvent e)
{
l.setText("Key Released");
}
public void keyTyped(KeyEvent e)
{
```

```
l.setText("Key Typed");

}

public static void main(String[] args)

{

new KeyListenerExample();

}

}
```

Output:



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.

Adapter Class	Listener Interface
ComponentAdapter	ComponentListner
ContainerAdapter	ContainerListner
FocusAdapter	FocusListner
KeyAdapter	KeyListner
MouseAdapter	MouseListner
MouseMotionAdapter	MouseMotionListner
WindowAdapter	WindowListner

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();

} }

}

}
```

Output :