Java Notes

Unit - 1

Java: java is an object-oriented programing language which is developed by James Gosling at sun microsystems in 1991. It was initially named Oak.

Java Buzzwords:

- ➔ Simple
- ➔ Platform independence
- ➔ Secure
- ➔ Portable
- ➔ Multi-threaded
- ➔ Robust
- ➔ Reach standard libraries
- ➔ Object oriented programing

Data Type

java datatype is the type of variable to store different types of data. In java there are 8 types od data types, byte, short, char, int, long, float, double and Boolean which are categories in 4 types.

- ➔ Integers (byte, short, int and long)
- ➔ Floating Point (float and double)
- ➔ Character (char)
- ➔ Boolean (true/false)

Variables

Variables in java are the basic unit of storage in which we can store different types of data. In simple term variables are the containers in java in which we can store different data with their different data types. Variables have their own scope and lifetime by which we can define the visibility and the life of the variable.

- ➔ Defining variables: variables are defined by defining their data type and variable name.
  Syntex:
  <data type> <variable_name> [=value (optinal)]
- ➔ Initialization: can be assign after the defining the variable or at the time of defining it
  Syntex:
  <data_type> <variable_name> = <value/expression>
  <variable_name> = <value/expression> {when variable already defined>

Scope and Lifetime of Variable

Scope and lifetime of variable refer to the visibility and life of the variable to access them. Or we can say that it refers to the region in which the variable can be accessed. Java supports 3 types of variables:

- ➔ Local Variables: local variables are the types of variables which can be accessed anywhere in the program. i.e., defined in the methods, blocks, and constructor. Their visibility is within the block where they are initialized.
- ➔ Instance Variables: instance variables are the types of variables which are defined within the class but outside the methods. They can access within the class and the instance (object) of that class. Their lifetime is till the object is not dead.
- ➔ Class Variables (Static Variables): static variables are the type of variables which are initialized within the class and outside the methods, but they can only access by there class and there value remains same for every instance of that class and associated with the class.

Operators

In java, operators are the symbols which perform operation on operands, operands can be variable, literals or expressions. In java there are 7 types of operators, and you can memorize them by this code "CAAL BUT":

- ➔ Comparison Operators: It is a type of operator which is used to compare two different expressions. It consist of: <, >, <=, >=, !=
- ➔ Arithmetic Operators: These are type of operators which are used to perform basic mathematics operations which are: +, -, /, %, *
- ➔ Assignment Operators: these are the types of operators which are used to assign values to the variables, these are: =, -=, +=, /=, *=, %=, ^=, &=, |=, ~=, >>=, <<=, >>>=
- ➔ Logical Operators: These are the types of operators which are used to check the relation between two different expressions or used to combine two different expressions, it is also know as relational operators. It consists of &&, ||
- ➔ Bitwise Operators: These are the types of operators which are used to perform on bits instead of bytes, it consists of: &, |, ^, ~, >>, <<, >>>
- ➔ Unary Operators: These are the types of operators which are used to perfume on single variables, it consist of two operators only which are increment and decrement: ++, --
- ➔ Ternary Operators: These are the types of operators which is used to perform single line if else statement. Syntex: <var_name> = <condition > ? <statement> [true] : < <statement> [false]

Control Structures

Control structures in java are the type of structure which helps the developers to control the flow of program or execution. It consists of condition, looping and branching statements.

- ➔ Conditional Statements:
  - If statements
  - If else statements
  - If else if else statements
  - Switch statements
- ➔ Looping Statements:
  - For loop:

```
for (initialization; condition; update){
        // code to execute
}
```

  - While loop:

```
while(condition){
        // code to execute
}
```

  - Do while loop:

```
do{
        // code to execute
} while(condition)
```

  - For each loop:

```
for(type element: array){
        // code to execute
}
```

- ➔ Branching Statements:
  - Break
  - Continue
  - Return statements

Arrays

In java, arrays are the type of data structure which are used to store fixed-size sequential element of collection which have same data type. In this each element has their own unique index which helps in accessing the elements. This is the continent way to store and manipulate multiple values within a single variable.

- ➔ Array Declaration: array are declared using following syntax:
  - <data_type>[] <arr_name>;
  - <data_type> <arr_name>[];

➔ Array size Initialization: array size is initialized in many ways:

- Method 1:

```
>> Int[]num;
>> num = new int[<size>];
```

- Method 2:

```
>> int[]num = new int[10];
```

➔ Array values initialization: arrays values are also initialized in many ways

- Method 1:

```
>> int[] num = new int[] {1, 2, 3, 4, 5};
>> int[] num = {1, 2, 3, 4, 5};
```

- Method 2:

```
>> int[] num = new int[3];
>> num[0] = 1;
>> num[1] = 2;
>> num[2] = 3;
```

➔ Accessing Array elements: array elements can be accessed by their unique index

```
>> var = num[0];
>> System.out.println(var);
```

➔ Multi-dimensional Arrays: java supports multi-dimensional arrays which are array of arrays

- 2-D array:

```
>>int[] num = {
        {1, 2, 3},
        {4, 5, 6}
    }
```

- 3-D array:

```
>>int[] num = {
        {{1, 2, 3},

         {5, 6, 7}},

        {{7, 8, 9},

         {10, 11, 12}}

        }
```

Java Architecture

Java architecture is the architecture which is used to create and run java programs. It consists of 3 components, JDK (Java Development Kit), JVM (Java Virtual Machine), JRE (Java Runtime Environment)

➔ JDK: It stands for Java Development Kit (JDK)

- JDK is a software development kit which is used to develop java software applications.
- JDK provides tools and libraries to run and develop java programs.

- JDK consists of:
  - Javac: Java compiler, it is used to compile the java code to the java byte code.
  - Java: Java interpreter, it is used to execute the java byte code in JVM.
  - Javap: Java dissembler, it is used to examine the byte code instructions.
  - Javadoc: Documentation generator, generate the document for the java program
  - Debugger: tool for debugging the java application.

➔ JVM: It stands for Java Virtual Machine (JVM)
- JVM is used to compile the java code to the java byte code to run the java program
- It is used to run the java program into any machine
- It consists of:
  - Memory management: it is used to mange memory or to perform garbase collection auto magically.
  - Used to debug the java byte code
  - Also act as an interpreter to convert the java byte code to the machine native language.

➔ JRE: It stands for Java Runtime Environment (JRE)
- JRE is the subset of JDK
- JRE provides those tools and libraries which are use to run the java program in end user local machine.
- It consists of:
  - JRE consist of JVM to run the java program
  - JRE have tools and libraries to create a runtime environment to run java program platform independently.

In summary JDK is used to develop and compile the java code, JVM is used to provide a runtime environment and JRE is used to run the Java program in the end used application. By combining all of them it create the Java architecture.

Difference between Java and C++

Basis	Java	C++
Language Type	Object-oriented programing language.	Multi-paradigm programing language
Platform	Can run on any platform by the help of JVM and convert the code into bytecode.	It is platform dependent, code compile into native machine code.
Multiple Inheritance	Can not perform multiple inheritance directly.	Can perform multiple inheritance.
Multi-Threading	Supports multi-threading using thread class.	No concept of multi-threading.
Memory Management	Memory management done automatically by garbage collector	Memory management done manually by the help of pointers.
Pointers	Not support pointers	Supports pointer

Class: Properties, Methods and Constructors

When creating a class in java, you define properties (fields), methods and constructors. Here how you can create a simple class using properties, methods and constructors:

```
>> class MyClass{

        // ================= Properties ================
        public myInt;
        private myString;
        // ================= Constructors ==============
        public MyClass(){  // default constructor
                myInt=0;
                myString="";
        }
        Public MyClass(int num, String str){  //  parametrized constructor
                myInt = num;
                myStr = str;
        }
        Public Myclass(&Myclass obj){  // copy constructor
                myInt = obj.myInt;
                myStr = obj.myStr;
        }
        // ================= Methods =================
        public void setValues(int num, String str){
                myInt = num;
                myStr = str;
        }
        Public getValues(){
                System.out.println("myInt = " + myInt);
                System.out.println("myStr = "+ myStr);
        }
}
```

1). Properties (fields): properties represents the data member or attributes of the class. In the above example myInt and myStr are 2 different properties having different access specifiers.

2). Constructors: Constructors are the special type of methods which are used to initialize the properties of the class at the time of object creation. There are 3 types of constructors:

- ➔ Default Constructors: default constructors are the type of constructor which are called automatically and assign default values to the properties.
- ➔ Parameterized Constructors: these are the type of construct which have number of parameters by which assigns the values to the properties.

- ➔ Copy Constructors: These are the type of constructor which uses other object to assign the values the current object.

3). Methods: Methods are the behavior of the class which defines the function to execute by the class. In above example there are 2 methods, one is setValues which is used to assign values to the properties and other is getValues to display the properties or to show the values of the properties.

Object Access Modifiers

In java there are different types of object access modifiers which are used to define the access visibility of the variables, methods and constructors of the class. There are mainly four types of object access modifiers in java:

- ➔ Public: it is the type of access modifiers which can be access form anywhere in the program. i.e., it can be accessed within the class, outside the class with the help of object and by the inherited class.
- ➔ Private: It is the type of access modifiers which can only access within the class. i.e., it cannot access by the outside the class with the help of object and can not accessed by the inherited class, full secure.
- ➔ Protected: It is the type of access modifiers which can not access by the outside the class with the help object. i.e., it can access be accessed within the class and by the inherited class.
- ➔ Default: It is the special type of accessed modifiers where we don't have to assign any access modifiers, it can access only within that package. i.e., it cannot be accessed by other packages.

Method Overloading

Method overloading is the concept of polymorphism which helps in creating multiple methods with same name but different parameters. It is used when we want to create multiple similar functions but having different inputs.

Example:

```
>> public int sum(int num1, int num2){
        return num1+num2;
    }
    public int sum(int num1, int num2, int num2{
        return num1+num2+num3;
   }
   public double sum(double num1, double num2, double num3){
        return num1+num2+num3;
   }
```

Garbage Collection

Garbage collection in java is a process to deallocate unused memory and allocate that deallocated memory to the other element. It prevents memory leak by automatically detect unused elements and then deallocated the allocated memory. Garbage Collection is performed by Java Virtual Machine at the time of converting the java code into the java byte code.

this Keyword

In java, this keyword is a special type of keyword which can be used within the class and used to access the properties and methods of the class. The main use of this keyword is also used to distinguish between the actual argument and formal argument. Uses of this keyword:

➔ **1). Accessing instance elements**

```
>> class Myclass{
        int num;
        public Myclass(int num){
                this.num = num;
        }
    }
```

➔ **2). Invoking other Constructors**

```
>> class Myclass{
        int num1, num2;
        public Myclass(int num1){
                this.num1 = num1;
        }
        public Myclass(int num1, int num2){
                this(num1);
                this.num2 = num2;
        }
   }
```

➔ **3). Passing current object as a parameter**

```
>> class Myclass{
        public void method(){
                anotherMethod(this);
        }
        public void anotherMethod(Myclass obj){
                // method implementation
        }
   }
```

➔ **4). Returning Current Object**

```
>> public class MyClass {
         public MyClass getSelf() {
                 return this; // Return the current object
         }
    }
```

➔ **5). Chaining Method call**

```
>> public class Myclass{
        public Myclass method1(){
                // method implementation
                Return this;
```

```
        }
        public Myclass method2(){
                // method implementation
                Return this;
        }
 }
```

Static (Variable, Method, and Block)

In java, static is used to define properties and methods to associate them with the class rather than the instance of the class. Those properties and methods which are declared as static are shared same for each instance of the class and can be accessed directly with the class name.

- **Static Variable:** static variable is a variable which is declared within the class but outside the method using the static keyword. Static variables can only be accessed by the class name and the data of the static variable shared same to each instance of the class.

```
>> public class MyClass {
        public static int count = 0; // Static variable
        public MyClass() {
                count++; // Increment count each time a new instance is created
        }
 }
```

- **Static Method:** static method is a method which is declared with the static keyword and associated with the class rather than with instance of the class. It can have and manipulate only static variables and can call static functions.

```
>> public class MyClass {
        public static void printMessage() {
                System.out.println("Hello, world!");
        }
 }
```

- **Static Block:** static initialization blocks are the type of blocks where we can define static variables and can perform other static operations. They are executed once when the class if firstly loaded.

```
>> public class MyClass {
        public static int x;
        static {
                x = 10;
                System.out.println("Static block initialized");
        }
 }
```

final Keyword

In Java, the final keyword is a modifier that can be applied to classes, methods, variables, and parameters to show that they cannot be modified, overridden, or extended, depending on where it is used. Final keyword is used for security and optimization of the program.

- **Final Variables:** final variables are the variables which are once initialized then it cannot be modified or change. it uses final keyword to create final variable.

```
>> public class MyClass {
        final int x = 10; // Final variable
        public void printX() {
          System.out.println(x); // Read-only, cannot be modified
        }
  }
```

- **Final Method:** final method is a method which is once define final then it can not be override by any other inherit class.

```
>> public class Myclass{
        final void finalMethod(){
                // implementaion
        }
  }
public class YourClass extends Myclass{
        // implementation
        // cannot override the finalMethod()
}
```

- **Final Class:** final class is a type of class which is created by using final keyword which makes class not inheritable. i.e., final class cannot be inherited by other class.

```
>> final public class Myclass{
        // implementation
}
// cannot inherited by other sub class
```

- **Final Parameters:** final parameters are the type of parameters which are once defined as final then the value of that parameter cannot be change inside that method.

```
>> public class Myclass{
        public void process(final int x){
                // x = 10, Error: Cannot assign a value to final parameter
                System.out.println(x);
        }
  }
```

Wrapper Class

Wrapper class is a type of class in java which encapsulate the primitive type within the object. By the help of these wrapper class we can perform more functions/methods on the variables. Or we can say that, Wrapper class is a way to provide primitive type as an object so that we can use more utility function like converting, comparisons, manipulating primitive values.

- Short, Byte, Integer
- Long, Character, Float
- Double, Boolean

Usage of Wrapper Class

- ➔ Converting primitives to objects: Integer num = Integer.valueOf(5);
- ➔ Converting Objects to primitives: int num2 = num.intValue();
- ➔ Autoboxing and Unboxing: Integer num = 5;
  int num2 = num;
- ➔ Utility Methods: int parsedInt = Integer.parseInt("123");
- ➔ Null ability: Integer num = null;
- ➔ Can work with Collections: ArrayList<Integer> num = new ArrayList<>();

String Class and Methods

In Java, the String class represents a sequence of characters. It is widely used for storing and manipulating text-based data in Java programs. The String class is immutable, meaning that once a String object is created, its value cannot be changed. However, you can create new String objects with modified values.

Methods:

String str = "Hello World";

- ➔ Length: int num = str.length();
- ➔ charAt: char c = str.charAt(3);
- ➔ UpperCase: String upStr = str.toUppeCase();
- ➔ LowerCase: String lowStr = str.toLowerCase();
- ➔ indexOf: int num = str.indexOf('W');

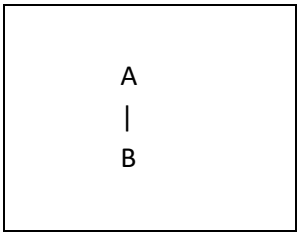
Unit – 2

Inheritance

In java, inheritance is a concept by which one class can inherit the properties (fields) and methods(behaviors) of other class. The class which inherits the properties and method are called as sub class and from which it inherits the properties and methods are called as super class. It helps In code modularity and readability. Types of Inheritance:

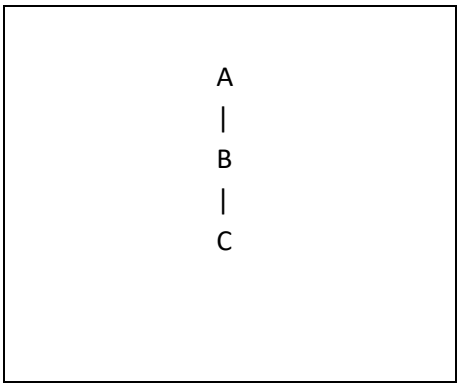
- **Single Inheritance:** It is a type of inheritance in which a single class inherits the other class only. Here only one class derived from other class.

```
>> class A{
        int a;
   }
   class B extends A{
        int b;
        public void display(){
            System.out.println("a = "+a);
            System.out.println("b = "+b);
        }
   }
```



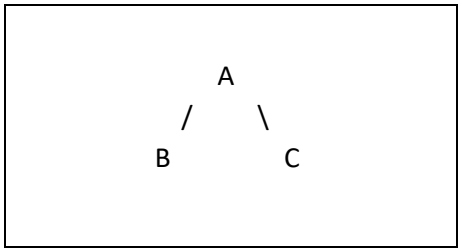
- **Multi-Level Inheritance:** It is a type of inheritance in which one class derived from other sub class. Here inheritance is done by level by level.

```
>> class A{
        int a;
   }
   class B extends A{
        int b;
   }
   class C extends B{
        int c;
        public void display(){
            System.out.println("A = "+a);
            System.out.println("B = "+b);
            System.out.println("C = "+c);
        }
   }
```



- **Hierarchal Inheritance:** It is a type of inheritance which creates a tree like structure where multiple sub class inherit the properties and method from single super class.

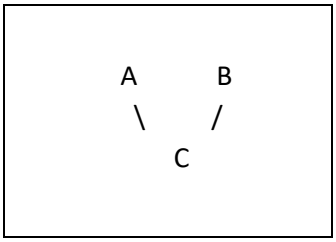
```
>> class A{
        int a;
   }
   class B extends A{
        int b;
```



```
        public void display(){
            System.out.println("A = "+a);
            System.out.println("B = "+b);
        }
}
class C extends A{
        int c;
        public void display(){
            System.out.println("A = "+a);
            System.out.println("B = "+b);
        }
}
```

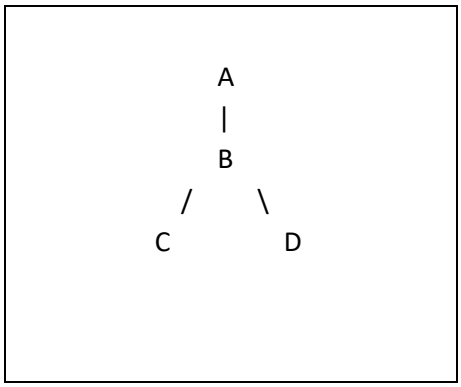
➔ **Multiple Inheritance:** It is a type of inheritance where a single sub class can inherit the properties and methods of two or more super class. In java, multiple inheritance cannot be implemented directly, for it we must use the concept of interface.

```
>> interface A{
        void displayA();
}
interface B{
        void displayB();
}
class C implements A, B{
        void displayA(){
            System.out.println("Class A");
        }
        Void displayB(){
            System.out.println("Class B");
        }
}
```



➔ **Hybrid Inheritance:** It is a type of inheritance which is the combination of other two or more type of inheritance except the multiple inheritance.

```
>> class A{
        int a;
} class B extends A{
        int b;
} class C extends B{
        int c;
}class D extends B{
        int d;
}
```



Super Keyword

In java, super keyword is a special type of keyword which is used to refer to the super class properties, methods and constructors within the sub class. super used for calling super class variables, methods, and constructors within the sub class.

- **Accessing super class Members:** You can access the super class members (fields or properties) using the 'super' keyword within the subclass. It is helpful when you override a member but you still want to access it.
- **Accessing super class Constructors:** You can access the super class constructor by using the 'super' keyword but in the subclass construct first you have to call the super class.

```
>> class A{
        int numA;
        A(int num){
                this.numA = num;
        }
        void display(){
                Systemlout.println("numA = "+numA);
        }
}
class B extends A{
        int numb;
        B(int a, int b){
                super(a);   // Calling Constructor
                this.numB = b;
        }
        void display(){
                super.display();  // Calling Method and Can also call Properties
                System.out.println("numb = "+numB);
        }
}
```

Method Overriding

Method overriding is a polymorphism concept in java by which we can re-define already written method of super class inside the sub class. It is use full when multiple subclass inherit the super class and using methods in there different way.

```
>> class Square{
        int side;
        Circle(int side){
                this.side = side;
        }
        public void area(){
                System.out.println("area =" + side*side);
        }
```

```
}
class Rectangle extends Square{
        int l, b;
        Rectangle(int l, int b){
                this.l = l;
                this.b = b;
        }
        public void area(){
                System.out.println("area = " + l*b);
        }
}
```

Key points:

- ➔ Method signature should be same in both super call as sub class.
- ➔ Return type should also be same.
- ➔ Access modifiers in sub class should be same or less restricted.
- ➔ Methods that re static, final or private cannot be overridden.

Covariant Return Type

Covariant return type is the special type of return type in java by which we can change the return type of the overridden method to the subtype of return type of super class method. This concept introduce in java 5.0 version.

Generally we matches the signature of super class method with subclass method include its return type but with the help of covariant return type we can change the sub class return type with the sub type return type of the super class.

Example: >>

```
class A{
        public A returnClass(){
                return new A();
        }
} class B extends A{
        public B returnClass(){
                return new B();
        }
}
```

Abstract Class

Abstract class is a class in java which object never created and it consist of concrete methods and abstract methods which are than implemented inside the subclass. Abstract class can have properties and method with constructors.

Abstract class are designed for extend by subclass to provide the concrete implementation of abstract methods.

```
>> abstract class Shape{
        abstract double area();  // abstract method
        void display(){ // concrete method
                System.out.println("This is a Shape");
        }
}
class Circle extends Shape{
        double radius;
        Circle(double radius){
                this.radius = radius;
        }

        double area(){ // concrete implementation of abstract method
                return Math.pi * radius * radius;
        }
}
```

Interface

In java, interface is a special type of class which is used to implement the multiple inheritance. Interface consist of only abstract methods. I.e., it cannot have any concrete method or properties and cannot have its constructor.

- ➔ **Creating Interface:** to create the interface we uses 'interface' keyword followed by the interface name. inside the interface we define different abstract method without there concrete implementation
- ➔ **Implementing Interface:** to implement the interface we uses 'implements' keyword instead of extends. Inside the subclass then we define the concrete implementation of all the abstract method of the interface

  Example:

```
>> interface Animal{
        void makeSound();
        void move();
}
Class Dog implements Animal{
        Void makeSound(){
                System.out.println("Bow Bow Bow !!!");
        }
        Void move(){
                System.out.println("Running on four legs");
        }
}
```

```
public class Myclass{
        public static void main(String args[]){
                Dog d = new Dog();
                d.makeSound();
                d.move();
        }
}
```

Difference Between Abstract class And Interface

Basis	**Abstract Class**	**Interface**
Concrete Methods	Abstract class can have both concrete and abstract methods	Interface can only have abstract methods
Constructors	Abstract class can have constructors to assign values to the properties	Interface does not supports constructors
Properties	Abstract class can have properties	Interface don't have properties
Multiple Inheritance	Abstract class does not support multiple inheritance as it has concrete methods as well	Interface supports multiple inheritance with the help of abstract methods
implementation	Abstract classes are implemented using extends keyword	Interfaces are implemented using implements keyword.
Example	Consider above examples	Consider above examples

Packages

Packages in java is a process of combining the different modules and classes into a single unit. It is used for differentiating naming convention and used to store the same type of modules into a single package. We can define packages by our self as well as java provides reach libraries and packages to use in the java program.

- **Creating Package:** to create the package, we have to define the package name at the top of the java program file with package keyword.
    - >> package com.example.myClass

```
class A{
        int a;
        A(int a){
                this.a = a;
        }
}
```

- **Importing package:** to import the package into the another file we can use import statement followed by the package name. we can import particular class or import all class

```
>> import com.example.myClass.A  // import only class A or,
Import com.explample.myClass.* // import all classes
```

Exception Class

Exception class is a class by which is used to handle exceptions in the java program. Exception class provides different – different checked and unchecked exceptions to handle the errors. With the help of Exception class user can also create there own exception class by extending the exception class an creating the constructor and calling the constructor of the super class by using super keyword.

```
>> public class MyClass {
   public static void main(String[] args) {
      int num1 = 10, num2 = 0;
      try {
         int res = num1 / num2;
      } catch (ArithmeticException e) {
         System.out.println("Cannot divide number by zero: " + e.getMessage());
      }
   }
}
```

➔ Creating a Custom Exception

```
>> public class CustomException extends Exception{
            CustomException(){
                  super("It is a customException")
            }
            CustomException(String message){
                  super(message);
            }
      } public class MyClass {
         public static void main(String[] args) {
            try {
               throw new CoustomException();
            } catch (CustomException e) {
               System.out.println("cought a Exception: " + e.getMessage());
            }
         }
      }
```

Built-In Checked and Unchecked Exceptions

Built-in Checked and Unchecked Exceptions are the 2 types of exceptions which are provided by the java Exception class.

➔ **Checked Exceptions:** checked exceptions are the types of exception that are checked at compile time. It should be caught by try and catch block explicitly. Or in simple terms we can say that it is a type of exception which should be explicitly handle by the user and cheked at the compilation time.

Example:

- **IOException**
- **FileInputOutputException**
- **SQLException**

➔ **Unchecked Exceptions:** These are the exceptions that are not checked at compile time. so it is not forced by the compiler's to either handle or specify the exception.

Examples:

- ArithmeticException
- NullPointerException
- ArrayIndexOutOfOboundsException

User Defined Exceptions / Use of try, catch, throw, throws and finally

User defined Exceptions are the custom exceptions which are created by the user using Exception Class. these exceptions extends the Exception class and then call the constructor of the Exception class inside the custom exception class constructor. It makes our application more readable and simpler as we can define our own custom exception according to the problem

Example:

```
class InsufficientBalanceException extends Exception{
        InsufficientBalanceException(){
                super("Your balance is insufficient");
        }
        InsufficientBalanceException(String message){
                super(message);
        }
}
class Bank{
        int balance;
        bank(int balance){
                this.balance=balance;
        }
        void deposite(int money){
                this.balance += money;
        }
        void widraw(int amount) throws InsufficientBalanceException{
                if amount > this.balance{
                        throw InsufficientBalanceException();
                }
        }
}
```

```
class Myclass{
	public static void main(String args[]){
		try{
			Bank parson = new Bank(100);
			Person.deposit(400);
			Person.widraw(600);
			Person.deposite(100);
		}
		Catch(InsufficientBalanceException e){
			System.out.println(“Insufficient Balance: ”+e.getMessage());
		}
		finally{
			System.out.println(“Thankyou for using our Bank 😊”);
		}
	}
}
```

Unit – 3

 # Multithreading

In java, multithreading is the concept by which we can execute two or more than two processes or program simultaneously. Multithreading is the most important concept in java as when we want to execute two processes simultaneously with one CPU then we use the concept of multithreading.

 # Multitasking

It is the process of executing multiple tasks simultaneously. For example, when we use chrome browser as well as listening to music, these are 2 tasks and they run simultaneously.

- ➔ Process-Based Multitasking: it is a type of multi-tasking where two or more independent tasks execute or run simultaneously. Example: using chrome while listening to a song.
- ➔ Thread-Based Multitasking: It is the type of multitasking where one process divided into sub process and these sub process execute or run simultaneously. Example: Using text editor print and edit text operation simultaneously to see the changes in the text.
- ➔ Difference:

➔	Heavyweight: requires more tools	➔	Lightweight: not requires other tools
➔	Own memory address space	➔	Share same memory space
➔	Inter-process comm. Is expensive	➔	Not that expensive as share same tools

Advantages of Multithreading

1) Can execute multiple thread simultaneously using single CPU.
2) Fast Execution due to the parallel processing.
3) Improve performance and concurrency.
4) Simultaneously access to multiple applications.

 # Thread Lifecycle

In java, thread is a sub program or module of java program. Thread lifecycle refer to how the thread appears in the following state until it terminated. A thread exists in any one of the following states. These states are:

1) New state
2) Active state
    a. Runnable
    b. Running
3) Blocked or Waiting state
4) Timed Waiting state
5) Terminated state

- ➔ **New state:** Whenever a thread is created, it goes into New state. The code which is in new state means that it is not running and executing.

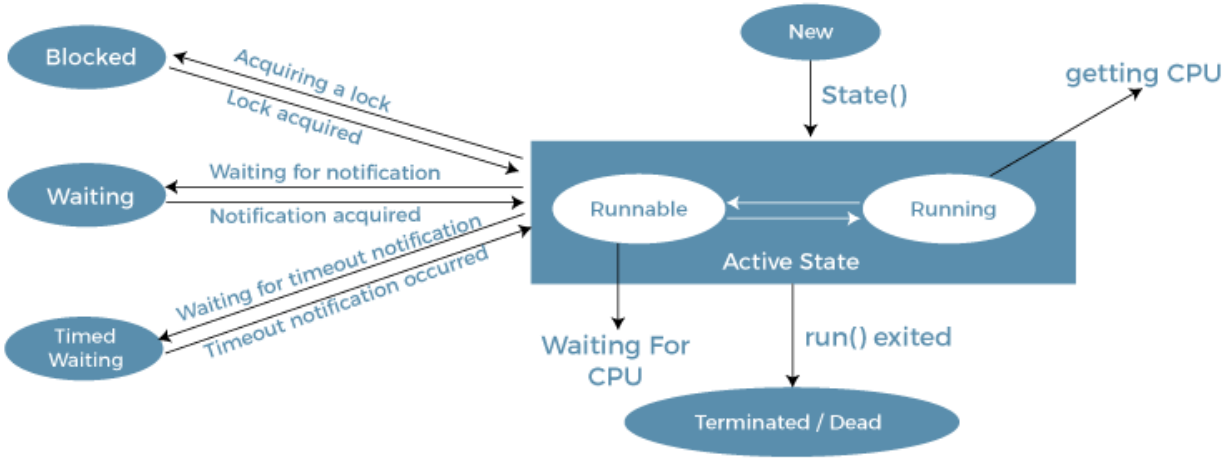
➔ **Active State:** When a thread invokes the start() method, it goes into active state. A thread which is in active state means that the thread is running or it may be run after some time when CPU is allocated to that thread. It consists of two other states:

- **Running:** It means that the current thread is executing or running. It goes into running state whenever thread get access to the CPU.
- **Runnable:** It means that current thread is ready to run and goes into Running state when it get access to the CPU. If we want to transfer a thread from Runnable state to Running state we uses yield() method.

➔ **Blocked or Waiting state:** Whenever a thread stop execution for a span period of time then it means that whether the thread is in blocked state or in waiting state. To transfer from active state to blocked/Waiting state we uses either join() or wait() method and again to transfer from bloacked/waiting state to active state we uses either notify() or notifyall() method.

➔ **Timed Waiting state:** whenever the thread calls the sleep(<period>) method, it goes into timed waiting state. A times waiting state is a state where the thread stop execution for certain period of time. And start execution when the time completes.

➔ **Terminated state:** Whenever thread calls the stop() method, the thread goes into terminated state. A terminated state is a state where the thread stop execution. The thread considered as dead and we have to again start the new thread.



Life Cycle of a Thread

 # Thread Creation in Java

Main thread: Every time a java program starts up, one thread begins running which is called as the main thread of the program because it is the one that is executed when you program begins.

We can create thread in two ways:

- ➔ **thread class:** first way to create a thread is to extend the thread class and write your program by implementing the run method. Example:

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

- ➔ **Runnable interface:** the second way to create a thread is to implements the Runnable interface and write your program by implementing the run method.

```
>> class A implements Runnable{
        public void run(){
                System.out.println("Thread A is running");
        }
        public static void main(String args[]){
                A objA = new A();
                Thread objAThread = new Thread(objA);
                objAThread.start();
        }
}
```

 # Creation of Multiple Thread

We can create multiple thread by extending the thread class in two different classes. An then we can run them simultaneously. Example:

```
>> class A extends Thread{
  public void run(){
    for(int i = 0; i<5; i++){
      System.out.println("ThreadA = "+i);
    }
    System.out.println("Exit from ThreadA");
  }
```

```
}
class B extends Thread{
   public void run(){
      for(int i = 0; i<5; i++){
         System.out.println("ThreadB = "+i);
      }
      System.out.println("Exit from ThreadB");
   }
}
public class Myclass{
   public static void main(String args[]){
      A objA = new A();
      B objB = new B();
      objA.start();
      objB.start();
   }
}
```

 # Implementation of Thread method

- ➔ **start():** start method is used to transfer from New state to Active State.
- ➔ **run():** It is the method in which we specifies the program that we want to execute.
- ➔ **yield():** It is used to transfer the thread from running state to runnable state.
- ➔ **stop():** stop method is used to kill the thread.
- ➔ **sleep(period)**: Sleep method is used to pause the execution of the thread for some period.
- ➔ **isAlive()**: Used to check if the thread is alive or not; it returns True or False.
- ➔ **join()**: Used to block the calling thread until the current thread completes.
- ➔ **wait()**: Used with condition variables to block the current thread until notified or until a timeout occurs.
- ➔ **notify()**: Used with condition variables to wake up one waiting thread.
- ➔ **notifyAll()**: Used with condition variables to wake up all waiting threads.

Example:

```
>> public class Myclass{

    public static void main(String args[]){
        A objA = new A();
        B objB = new B();
        objA.start();
        objB.start();

        objA.isAlive();
    }
}
```

```
class A extends Thread{
    public void run(){
        for(int i = 0; i<5; i++){
            try{
                if(i==2){
                    sleep(1000);
                }
                if(i == 3){
                    Thread.yield();
                }
                if(i == 4){
                    // join();
                    continue;
                }
            } catch(Exception e){
                System.out.println(e);
            }
            System.out.println("ThreadA = "+i);
        }
        System.out.println("Exit from ThreadA");
    }
}
class B extends Thread{
    public void run(){
        for(int i = 0; i<5; i++){
            try{
                if(i==2){
                    wait(100);
                }
                if(i == 3){
                    Thread.yield();
                }
                if(i == 4){
                    notifyAll();
                }
            } catch(Exception e){
                System.out.println(e);
            }
            System.out.println("ThreadB = "+i);
        }
        System.out.println("Exit from ThreadB");
    }
}
```

 # Synchronization

- ➔ Used to solve data inconsistency problem.
- ➔ Synchronization is only applicable in methods and block
- ➔ Synchronization block creates a block of code where all the code execute one after the other.
- ➔ Synchronization Method is used to synchronize the use of the method, if one thread is using that method tan other thread cannot access it until the thread one completes.

Example:

```
>> class A {
   synchronized void add(int n){
      Thread t = Thread.currentThread();
      for(int i = n; i<=5; i++){
         System.out.println(t.getName() + " : " + i);
      }
   }
}
class B extends Thread{
   A objA = new A();
   public void run(){
      objA.add(1);
   }
}
public class Myclass{
   public static void main(String args[]){
      B t = new B();
      Thread t1 = new Thread(t);
      Thread t2 = new Thread(t);
      t1.setName("T1");
      t2.setName("T2");
      t1.start();
      t2.start();
   }
}
```

 # Thread Priorities

Thread priorities is the concept by which we can assign a priority of execution of the thread. Here we have 3 priorities, min, max, norm

Min: 1

Norm: 5

Max: 10

If a thread has priority 10, then even if that thread calls yield method but still it executes it completely.

Example:

```
Thread.setPriority(0);
```

 # Elementary concepts of Input/Output

- ➔ Java I/O is used to process the input and produce the output.
- ➔ Java uses the concept of stream to make I/O fast.
- ➔ The java.io contains all the classes that are used to perform I/O operations.
- ➔ We can perform file handling in java by using Java API

Stream:

- ➔ Stream is a sequence of data
- ➔ In java, Stream is a sequence of bytes.
- ➔ It is called stream because it consist of sequences of elements.

In java, 3 streams are created automatically

1) System.in: Standard input stream
2) System.out: Standard output stream
3) System.err: Standard error stream

 # Using Byte Stream / Reading and Writing using byte

Byte streams are used for handling raw binary data in Java. They are particularly useful when dealing with files or network communication, where data is often represented as a sequence of bytes. Here's an overview of using byte streams for input and output:

- ➔ **FileInputStream:** For reading the byte streams, we can use FileInputStream to create it's object and perform the operation. It is necessary to use try, catch block as method throws and IOException. Some method used by File Input Stream:

Methods	**Description**
1) Available	Returns an estimate of the number of remaining bytes that can be read (or skipped over) from this input stream.
2) close	Closes this file input stream
3) finalize	Ensures that the close method of this file input stream is called when there are no more references to it.
4) read	Reads a byte of data from this input stream
5) skip	Skips over and discards n bytes of data from the input stream

Example:

```
import java.io.*;

public class file_input_Stream {
   public static void main(String args[]){
      try(FileInputStream fin = new FileInputStream("test.txt")){
         int i = 0;
         fin.skip(5);
         while((i = fin.read()) != -1){
            System.out.println((char)i + " Available bytes: " + fin.available());

         }
         fin.close(); // not required as finalize will automatically ensures it
      } catch(Exception e){
         System.out.println(e);
      }
   }
}
```

➔ **FileOutputStream:** For writing the byte stream, we can use FileOuputStream to create it's object and perform the operation. It is necessary to use try, catch block as method throws and IOException. Some method used by File Ouput Stream:

Method	**Description**
1) Write	Used to right byte stream
2) flush	For clearing the OutputStream, we use the flush() method. This method forces all the data to get stored to its destination.
3) Close	It closes the file output stream
4) Finalize	It is used to clean up all the connection with the file output stream and finalize the data.
5) write(byte[] arr)	It is used to write data in bytes of arr[] to file output stream.

Example:

```
Import java.io.*;
Public class file_output_stream{
        Public static void main(String args[]){
                Try(FileOutputStream fos = new FileOutputStream("test.txt")){
                        String text = "Hello, World!"
                        Byte[] b = text.getBytes();
                        fos.write(b);
                        System.out.println("file written Successfully");
```

```
            } catch(Exception e){
                    System.out.println(e)
            }
        }
}
```

 # Automatically closing a file

- ➔ In java, you can ensure that a file will automatically close when your work done or exited from try-catch block.
- ➔ It utilizes the concept of try-with-resources
- ➔ This concept introduces in java 7 to simplify the process.
- ➔ We can also use close method directly to close a file

 # Using Character-Based stream / Reading and Writing using Character stream

Character-based streams are used for handling text-based data in Java. They are particularly useful when dealing with text files or when you need to process textual data. Here's how you can use character-based streams for reading from and writing to files:

- ➔ **FileReader:** If we want to read the textual data from file, we uses FileReader. We create an object of FileReader and use read method to read the character stream from the file. It throws and IOException so it is required to use try catch statement. Some methods are:

Methods	**Descriptions**
1) Close	To close the file.
2) Mark	Marks the present position in the stream.
3) Reset	Reset the position at the beginning of file
4) Read	Read a 2 byte character
5) Skip()	Skip n number of character

Example:

```
import java.io.*;
public class file_reader {
   public static void main(String args[]){
      try(FileReader fr = new FileReader("test.txt")){
         int i = 0;
         fr.skip(5);
         while((i = fr.read()) != -1){
            System.out.print((char)i);
         }
         fr.close();
```

```
      } catch(Exception e){
         System.out.println(e);
      }
   }
}
```

➔ **FileWriter:** If we want to write the textual data in a file, we uses FileWriter. We create the object of FileWriter and uses write method to write character stream into the file. It throws IOException so it is required to use try catch block. Some methods use:

Method	**Description**
1) Append	Append a single character to the file
2) append(CharSequence char_sq)	appends specified character sequence to the writer
3) flush	Flush the WriterStream
4) write	Wite a single character to the stream
5) write(string str)	write a string to the character stream.

Example:

```
import java.io.*;
public class file_writer {
   public static void main(String args[]){
      try(FileWriter fw = new FileWriter("test.txt")){
         fw.write("Hello World");
         System.out.println("File written successfully");
      } catch(Exception e){
         System.out.println(e);
      }
   }
}
```

Unit – 4

 # Java Swing

- ➔ Java swing is a Java Foundation Class which is used to create Window-Based applications.
- ➔ It is built on top of the AWT (Abstract Window Toolkit) API and entirely written in java.
- ➔ Unlike AWT, java swing provides platform independent and light weight components.
- ➔ Javax.swing package provides classes for java swing API such as JBotton, JTextFeild, JRadioButton, JSlider, etc.

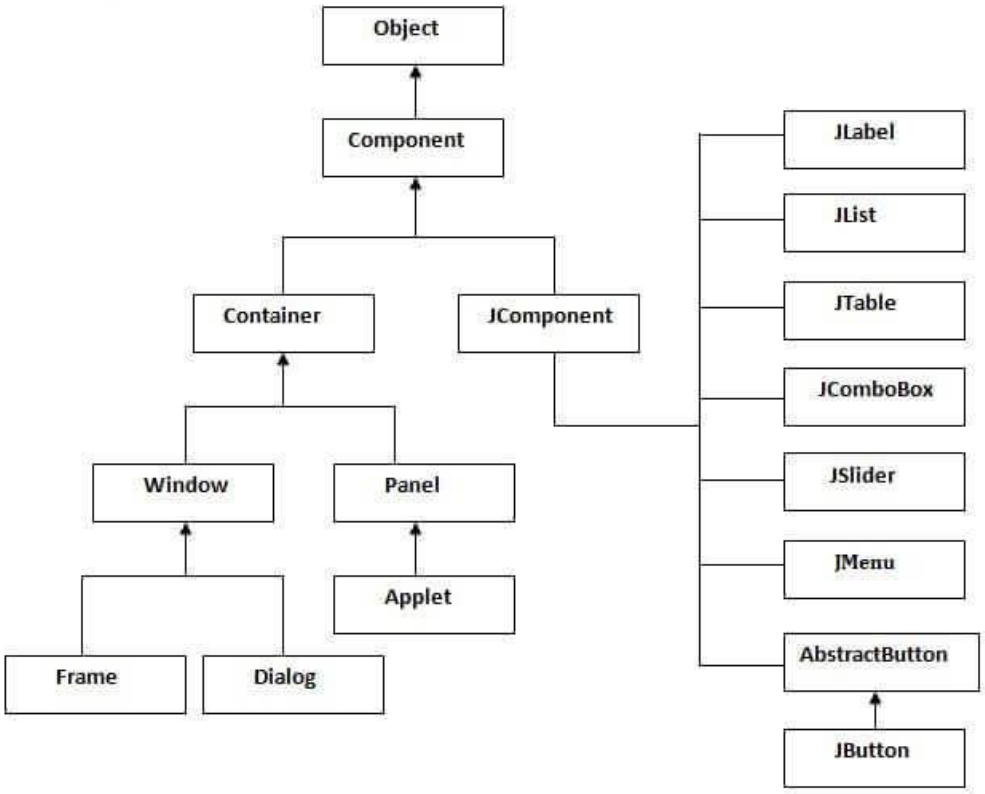
 # Difference between AWT and java swing

Java AWT	Java Swing
1). Java AWT are platform-dependent	1). They are platform-independent
2). AWT components are heavyweight	2). Swing components are lightweight
3). AWT doesn't support pluggable look and feel	3). Swing supports pluggable look and feel
4). AWT have less component as swing	4). Swing provides more powerful components.
5). AWT doesn't support Model View Controller (MVC)	5). Swing follows MVC.

 # Java Foundation Class (JFC)

The java foundation class are the GUI components which are used to develop desktop applications.

Hierarchy of Java Swing class:



 # commonly used method of components

Method	Description
1). public void add(Component c)	Add a component on another component.
2). Public void setSize(int width, int height)	Sets size of the components.
3). Public void setLeyout(LayoutManager m)	Sets the layout manager for the component
4). Public void setVisible(Boolean b)	Set visibility of the components. It's default False

Java Swing Example:

There are two ways to create Frames in java:

➔ **By creating the object of the Frame class**

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

public class swing{
   public static void main(String args[]){
      JFrame f = new JFrame("My Frame"); // creating object of JFrame
      JButton b = new JButton("Click"); // Creating a Object of JButton
      b.setBounds(130, 100, 100, 40); // seting the position of button
      f.add(b); // add button to JFrame

      f.setSize(400, 500);  // set size of window
      f.setLayout(null); // set the layout
      f.setVisible(true); // frame visibility
   }
}
```

➔ **By extending the Frame class**

```
>> import javax.swing.*;
public class Simple2 extends JFrame {// inheriting JFrame
   JFrame f;
   Simple2() {
      JButton b = new JButton("click");// create button
      b.setBounds(130, 100, 100, 40);

      add(b);// adding button on frame
      setSize(400, 500);
      setLayout(null);
      setVisible(true);
   }
   public static void main(String[] args) {
      new Simple2();
   }
}
```

 # Java Swing Components

There are various components tin java to create a simple desktop application, but some of them are:

1) **JLable:** The object of JLabel class is a component for placing text or image in a container. It is used to display a single line of read only text. It inherits JComponent class.
   Example:
   ```
   JLabel label = new JLabel("First Lable"); // creating a object of JLable
   label.setBounds(50, 50, 100, 30); // set the position of lable
   f.add(label); // add lable to JFrame
   ```

2) **ImageIcon:** It is a class in swing used to represent image icon. It is used to load image icon by specifying the source such as URL, Files, etc.
   Example:
   ```
   ImageIcon img = new ImageIcon("picture.PNG"); // creating a object of ImageIcon
   JLabel l = new JLabel(img); // creating a object of JLabel
   l.setBounds(50, 150, 100, 100); // set the position of lable
   f.add(l); // add lable to JFrame
   ```

3) JButton: JButton are the class in java used to create clickable buttons. We can use addActionListener method to perform events when clicked at the button. We can add text later using the setText() method or we can add text at the time of button creation. Used for triggering action. Example:
```
JButton b = new JButton("Click"); // Creating a Object of JButton
b.setBounds(130, 100, 100, 40); // seting the position of button
f.add(b); // add button to JFrame
```

4) JToggleButton: JToggleButton is a swing component which is used to hold two states, on and off. We can write the button name using the setText command. We can set the position, height and width. We can also attach ActionListener. Example:
```
JToggleButton tb = new JToggleButton("ON/OFF");
tb.setBounds(100, 250, 100, 30);
f.add(tb);
```

5) JCheckBox: JCheckBox is a swing component which is used to create check boxes using array of items. It represents small check box which can be checked or unchecked. Can set text using setText. Example:
```
JCheckBox item1 = new JCheckBox("item 1");
item1.setBounds(100, 300, 100, 30);
f.add(item1);
```

6) **JRadioButton:** JRadioButton is a swing component which is used to create radio button on GUI. It is a small circular button which can be selected or deselect by the user. We can add text using setText(). Use when only one option to select. Example:

```
JRadioButton r1 = new JRadioButton("A");
JRadioButton r2 = new JRadioButton("B");
r1.setBounds(100, 350, 50, 30); r2.setBounds(150, 350, 50, 30);
ButtonGroup bg = new ButtonGroup();
bg.add(r1); bg.add(r2);
f.add(r1);
f.add(r2);
```

7) **JTextField:** JTextField is a swing container which is used to create a text field on GUI. We can set the text using setText. It is used to create single line input text field. It allows user to input text from keyboard. Example:

```
JTextField t = new JTextField("Type Here");
t.setBounds(100, 400, 100, 30);
f.add(t);
```

8) **JScrollPane:** JScrollPane is a swing component which is used give scrolling capabilities to those components whose container area is too small to fit the component. It is comanly used for scrolling function and mostly use with JList or JTable which have large amount of data. Example:

```
JScrollPane sp = new JScrollPane();
sp.setBounds(100, 450, 100, 100);
sp.setViewportView(<component>);
f.add(sp);
```

9) **JList:** JList is a swing component which is used to create a list of items to display on GUI. You can create JLIst using the array which can be of any datatype with some elements or items. JList supports single or multiple selection, adding, removing, updating, etc.
Example:

```
String[] items = {"A", "B", "C", "D"};
JList<String> list = new JList<>(items);
list.setBounds(100, 550, 100, 100);
list.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
f.add(list);
```

10) **JComboBox:** JComboBox is a java swing component which is used to create a drop down menu on GUI to select one option. We can chose the selection mode. Example:

```
JComboBox<String> cb = new JComboBox<>(items);
cb.setBounds(100, 650, 100, 30);
f.add(cb);
```

11) **Containers:** In java, containers are the special type of swing components which is used to store and manage other containers. Container are used to layout and organize the UI components. Example: JFrame, JPanel, JScrollPane

 # Layout managers

- ➔ Layout managers are used to manage the visual layout of a container on GUI. It is used to set the position of components within a container.
- ➔ They provide some algorithm and methods to manage the components which are inside the containers.
- ➔ Swing provides different types of layout such as boarderLayout, flowLayout, etc and can be set using the setLayout().
- ➔ Example: Frame.setLayout(new GridLayout()); // set the layout

 # Event Delegation Model

1) In Java Swing, the event delegation model is a special type of model used to handle the different events occur in the swing components.
2) It generates events and then listener responses to the events.
3) We can add or remove event listener at runtime.

 # Event Handling

Java Swing's event handling mechanism is based on the event delegation model, which involves event sources, event listeners, event classes, and adapter classes. Here is an overview of each component:

- ➔ **Event Sources:** An event source is a component that generates events. Any Swing component can act as an event source (e.g., buttons, text fields, windows).

- ➔ **Event Listener:** An event listener is an object that listens for events from an event source. It must implement specific interfaces provided by the AWT (Abstract Window Toolkit) and Swing libraries. Example: ActionListener, MouseListener, KeyListener, etc.

- **Event Class:** Event classes encapsulate the details of an event. They extend the `java.util.EventObject` class. Example: ActionEvent, MouseEvent, KeyEvent, etc. These classes provide methods to retrieve event details, like the source of the event, the time it occurred, etc.

- **Adapter Class:** Adapter classes provide default implementations for listener interfaces with multiple methods. They allow developers to override only the methods they need, rather than implementing all methods of the interface.
    - `MouseAdapter`: Implements `MouseListener` and `MouseMotionListener` interfaces.
    - `KeyAdapter`: Implements `KeyListener` interface.

Example of Event Handling using Java Swing:

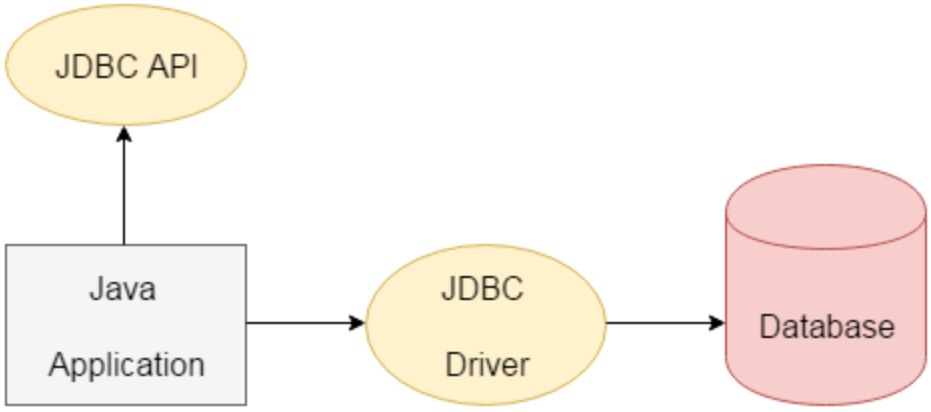
```
import javax.swing.*;
import java.awt.event.*;

public class MouseEventExample {

   public static void main(String[] args) {
      JFrame frame = new JFrame("Mouse Event Example");
      JButton bt = new JButton("Click Me");
      bt.setBounds(50, 100, 95, 30);

      ActionListener actionListener = new ActionListener() {
         public void actionPerformed(ActionEvent e){
            System.out.println("Action Performed");
         }
      };
      bt.addActionListener(actionListener);

      // Adding the panel to the frame
      frame.add(bt);
      frame.setSize(300, 300);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

Example using Adapter Class:

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

public class MouseEventExample {

   public static void main(String[] args) {
      JFrame frame = new JFrame("Mouse Event Example");
      JPanel panel = new JPanel();

      // Event Listener using MouseAdapter
      panel.addMouseListener(new MouseAdapter() {
         public void mouseClicked(MouseEvent e) {
            // Event Handling Code
            System.out.println("Mouse was clicked at (" + e.getX() + ", " + e.getY() + ")");
         }

         public void mouseEntered(MouseEvent e) {
            System.out.println("Mouse entered the panel");
         }
      });

      // Adding the panel to the frame
      frame.add(panel);
      frame.setSize(300, 300);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

 # JDBC Architecture

- ➔ JDBC stands for Java Database Connectivity.
- ➔ By the help of JDBC architecture we can perform operations with various database using java
- ➔ Basically, JDBC is a API to connect and execute the query with the database.
- ➔ JDBC architecture consist of 4 parts: Java API, Java Application, JDBC Driver and Database.
- ➔ **Java API:** By the help of JDBC API we can perform queries in database
- ➔ **Java Application:** Here we write our java program
- ➔ **JDBC Driver:** It helps Java Application to connect with the database
- ➔ **Database:** It can be any storage application where we can store our tabular data.



 # JDBC Driver

JDBC Driver is a Software which helps in connecting the java application with the Database. There are 4 types of Driver:

1) JDBC-ODBC bridge Driver
2) Native-API Driver (partially Java Driver)
3) Network Protocol Driver (fully Java driver)
4) Thin Driver (fully java driver)

➔ **JDBC-ODBC bridge Driver:** It is a type of driver which uses ODBC driver to connect with the database. The JDBC-ODBC bridge driver converts the JDBC method calls into ODBC function calls. This is now discouraged due to the Thin Driver.

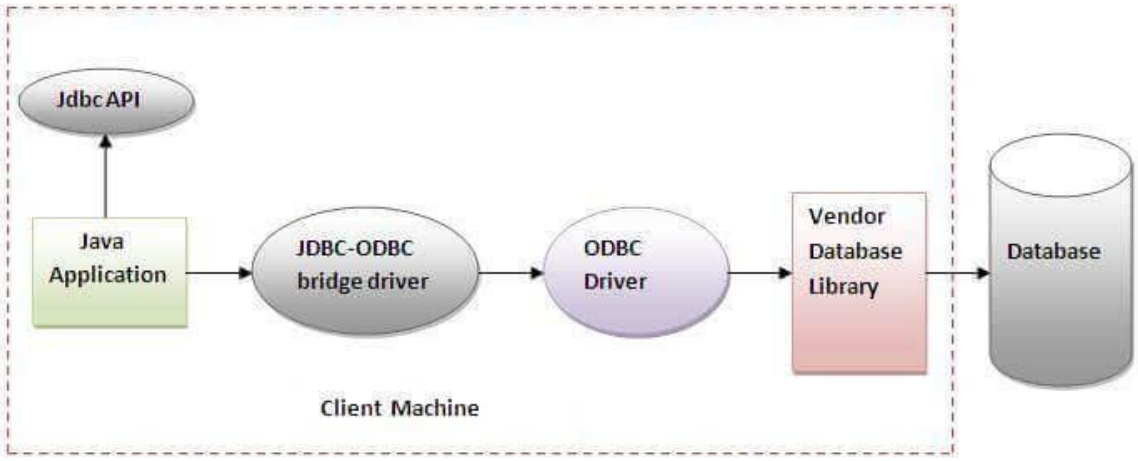


Figure- JDBC-ODBC Bridge Driver

➔ **Native-API Driver:** It is a type of driver which uses client-side libraries of database. The driver converts the JDBC calls into native calls of the database API. It is not written entirely in java.

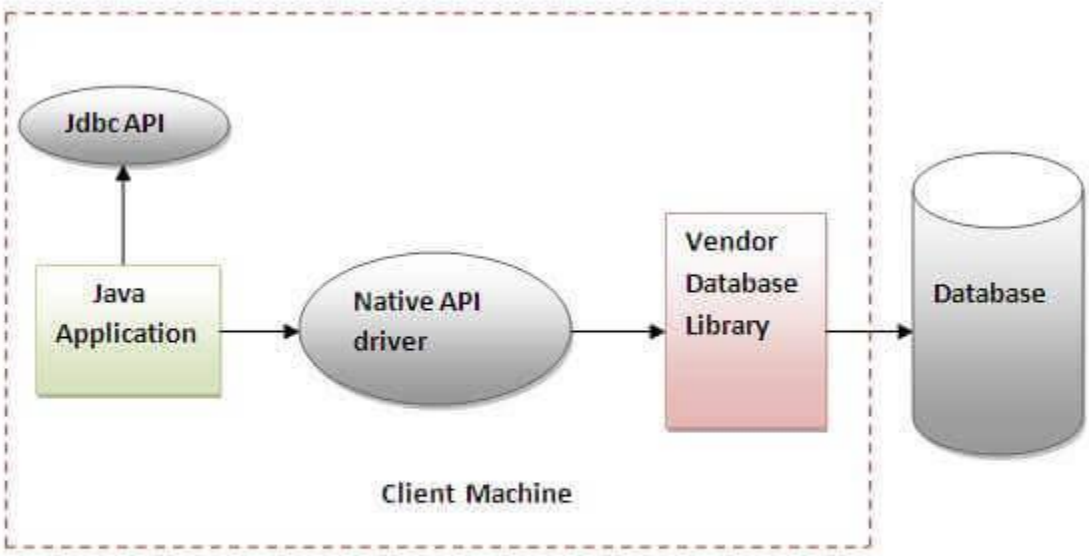


Figure- Native API Driver

➔ **Network Protocol Driver:** It is the type of JDBC Driver which uses middleware (application server) that convert the JDBC calls directly or indirectly onto the Vendor Specific Database protocol. It is fully written in java.

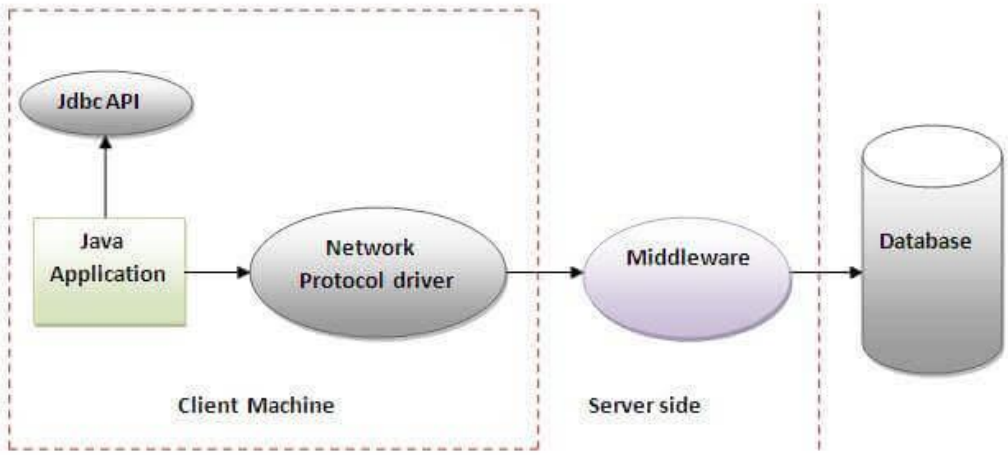


Figure- Network Protocol Driver

➔ **Thin Driver:** The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

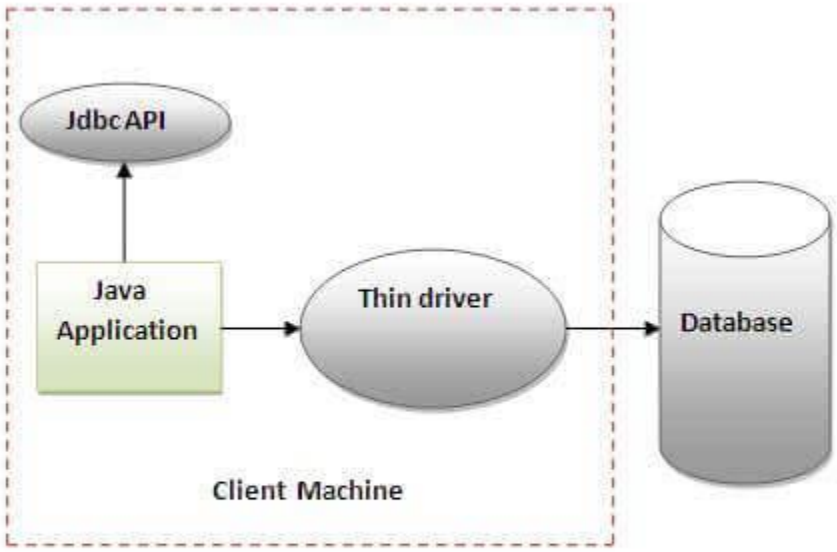


Figure- Thin Driver

 # Java Database Connectivity

There are 5 steps to connect with any java application with the database using JDBC. These steps are as follows:

1) Registering Driver Class
2) Creating Connection
3) Creating Statement
4) Executing Query
5) Close Connection

➔ **Registering Driver Class:** The forName() method of Class class is used to register the driver class. This method is used to dynamically load the driver class.
>> Class.forName("com.mysql.jdb.Driver");

➔ **Creating Connection:** The getConnection() method of DriverManger class is used to connect with the database. We must specify the path, usename, and password of DataBase.
>> Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3360/mydb", "root", "pass")

➔ **Creating Statement:** The createStatement() method of Connection class is used to create the Statement. The object of statement is responsible to execute the queries on database.
>> Statement st = con.createStatement();

➔ **Execute the Query:** The executeQuery() method of Statement object is used to execute the queries on the database. This method returns a ResultSet object that can be used to retrieve the data from the Database.
>> ResultSet rs = st.executeQuery(“Select * from Emp”);
While(rs.next()){
System.out.println(rs.getInt(1) + “ “ + rs.getString(2));
}

➔ **Close the Connection:** The close() method of Connection class is used to close the connection.
>> con.close();

Example:

```
>> import java.sql.*;
public class JDBCExam{
   public static void main(String args[]){
      try{ // Step - 1: Registering Driver Class
         Class.forName("com.mysql.jdbc.Driver");

         // Step - 2: Creating Connection
         Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3360/mydb", "root",
"pass");

         // Step - 3: Creating Statement
         Statement st = con.createStatement();

         // step - 4: Executing Queries
         ResultSet rs = st.executeQuery("select * from mydb");
         while(rs.next()){
            System.out.println(rs.getInt(1) + " " + rs.getString(2));
         }

         // Step - 5: Closing Connection
         con.close();
      } catch(Exception e){
         System.out.println(e);
      }
   }
}
```

 # Connection

Connection is an interface use to create a session between java application and database. It helps to establish a connection with the database.

Methods of Connection Interface:

>> createStatement(): use to create object of Statement class

>> autoCommit(): use to set the commit status. By default it is true.

>> commit(): save the changes made since the previous commit/rollback.

>> rollback(): Drops all change made since the previous commit/rollback.

>> close(): used to close the connection.

 # Statement

The createStatement() method of Connection class is used to create Statement. Statement helps in execting various querys including retrieve data, update data or remove data from database.

>> executeQuery(String sql): use to retrieve the data from the database table. Return ResultSet.

>> executeUpdate(String sql): use to execute query, it may be create, drop, insert, delete, etc. returns int.

>> execute(String sql): used to perform query that may return multiple results. Return Boolean.

>> executeBatch(): used to execute batch of command.

 # Prepared Statement

PreaparedStatement Interface is subInterface of Statement . It is used to execute parameterized queries. For example: String sql = “insert into emp values(?,?,?); As you can see, we are passing parameters (?) for the parameters. Its values are set by the setter method of PreparedStatement.

The preparedStatement of Connection class is used to create the object of Preapred Statement.

>> setInt(para_idx, value): set integer value to the specified index.

>> setString(para_idx, value): set String value to the specified index.

>> setFloat(para_idx, value): set Float value to the specified index.

>> executeUpdate(): executes the query. It is used to create, insert, update, delete, etc.

>> executeQuery(String sql): execute the select query. It return an instance of ResultSet.

We can use prepared statement when we want to perform a query at runtime.