

**Inheritance & Polymorphism****PART-1**

**Inheritance:** *Introduction, Forms of Inheritance - specialization, specification, construction, extension, limitation, combination, Member access rules, super keyword*  
**Polymorphism-** *method overriding, abstract classes, final keyword.*

## Inheritance

Inheritance is an important pillar of OOP (Object-Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class.

In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition, you can add new fields and methods to your current class as well.

The class which inherits the properties of other is known as **subclass** (derived class, child class) and the class whose properties are inherited is known as **superclass** (base class, parent class).

It represents a **parent-child relationship** between two classes. This parent-child relationship is also known as an **IS-A** relationship.

### Need of Inheritance

- **Code Reusability:** The basic need of an inheritance is to reuse the features. If you have defined some functionality once, by using the inheritance you can easily use them in other classes and packages.
- **Extensibility:** The inheritance helps to extend the functionalities of a class. If you have a base class with some functionalities, you can extend them by using the inheritance in the derived class.
- **Implementation of Method Overriding:** Inheritance is required to achieve one of the concepts of Polymorphism which is Method overriding.
- **Achieving Abstraction:** Another concept of OOPs that is abstraction also needs inheritance.

### Syntax to implement inheritance

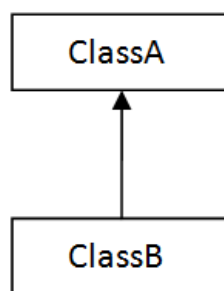
```
class Super
{
    ....
    ....
}
class Sub extends Super
{
    ....
    ....
}
```

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

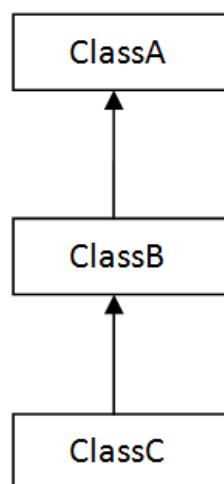
### Types of Inheritance in Java

Five different types of inheritances are possible in Object-Oriented Programming.

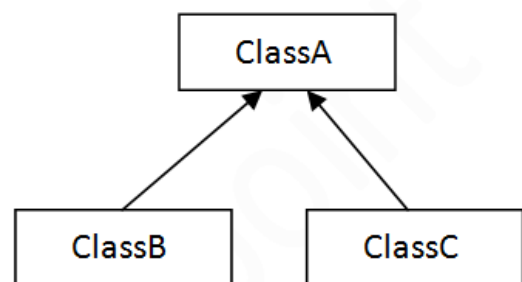
1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance



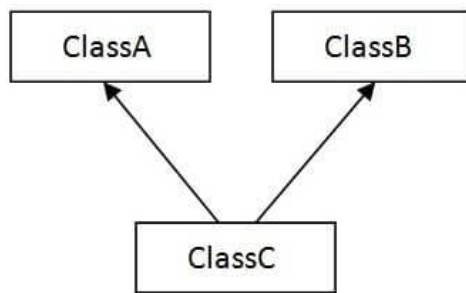
1) Single



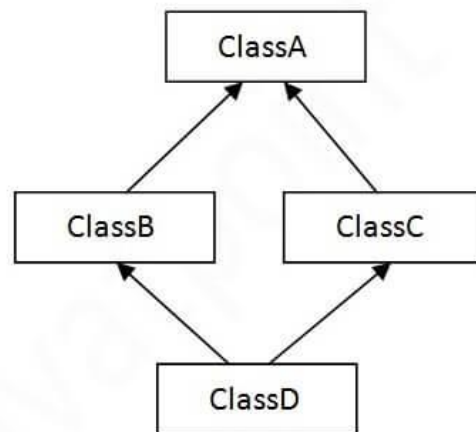
2) Multilevel



3) Hierarchical



4) Multiple



5) Hybrid

**Note:** Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For **Example:** there are three classes A, B, and C. The C class inherits A and B classes. If A and B classes have the same method and we call it from child class object, there will be ambiguity to call the method of A or B class.

### 1. Single Inheritance:

In single inheritance, a sub-class is derived from only one super class. It inherits the properties and behavior of a single-parent class. Sometimes, it is also known as simple inheritance. In the below figure, 'A' is a parent class and 'B' is a child class. The class 'B' inherits all the properties of the class 'A'.



Single Inheritance

*//Program on Single Inheritance*

*// Superclass: Vehicle*

class **Vehicle**

```
{
    String brand;
    int speed;

    public Vehicle(String brand, int speed)
    {
        this.brand = brand;
        this.speed = speed;
    }
    public void displayInfo()
    {
        System.out.println("Brand: " + brand);
        System.out.println("Speed: " + speed + " km/h");
    }
}
```

*// Subclass: Car*

class **Car extends Vehicle**

```
{
    int numberOfDoors;

    public Car(String brand, int speed, int numberOfDoors)
    {
        super(brand, speed);
        this.numberOfDoors = numberOfDoors;
    }

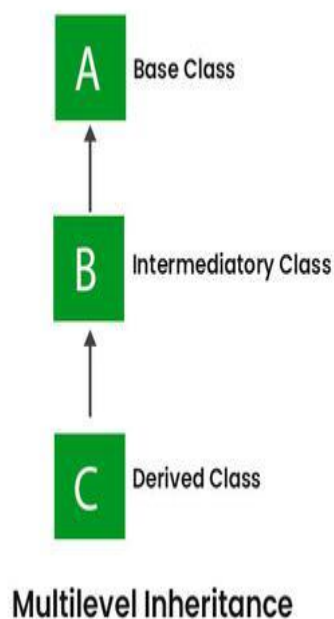
    public void displayCarInfo()
    {
        displayInfo();
        System.out.println("Number of Doors: " + numberOfDoors);
    }
}
```

```
public class Single
{
    public static void main(String[] args)
    {
        Car c = new Car("Toyota", 180, 4);
        c.displayCarInfo();
    }
}
```

```
F:\VJIT\JAVA\LAB\Unit-II\PART 1>javac Single.java
F:\VJIT\JAVA\LAB\Unit-II\PART 1>java Single
Brand: Toyota
Speed: 180 km/h
Number of Doors: 4
```

## 2. Multilevel Inheritance

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the **grandparent's members**.



*//Program on MultiLevel Inheritance*

*// Superclass: Vehicle*

```
class Vehicle
{
    String brand;
    public Vehicle(String brand)
    {
        this.brand = brand;
    }
    public void displayBrand()
    {
        System.out.println("Brand: " + brand);
    }
}
```

*// Subclass: Car (derived from Vehicle)*

```
class Car extends Vehicle
{
    int speed;
    public Car(String brand, int speed)
    {
        super(brand);
        this.speed = speed;
    }
    public void displaySpeed()
    {
        System.out.println("Speed: " + speed + " km/h");
    }
}
```

*// Subclass: SportsCar (derived from Car)*

```
class SportsCar extends Car
{
    boolean isTurbo;
    public SportsCar(String brand, int speed, boolean isTurbo)
    {
        super(brand, speed);
        this.isTurbo = isTurbo;
    }
}
```

```
public void displayTurbo()
{
    System.out.println("Turbo: " + (isTurbo ? "Yes" : "No"));
}
}
public class MultiLevel
{
    public static void main(String[] args)
    {
        SportsCar sc = new SportsCar("Ferrari", 350, true);
        sc.displayBrand();
        sc.displaySpeed();
        sc.displayTurbo();
    }
}
```

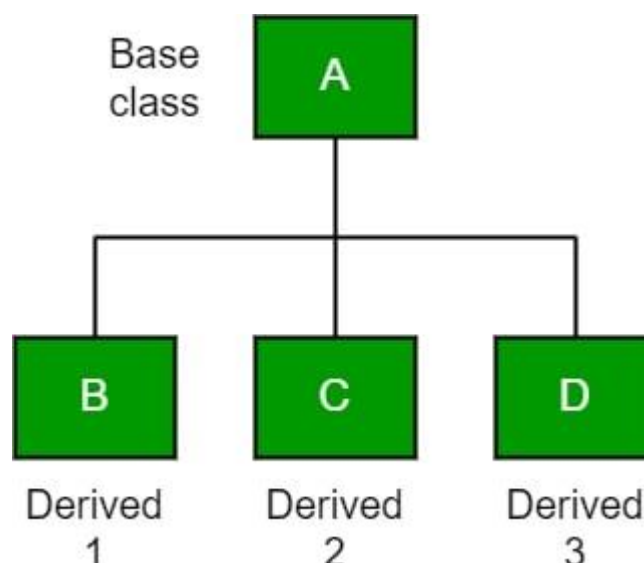
Command Prompt

```
F:\VJIT\JAVA\LAB\Unit-II\PART 1>java MultiLevel
Brand: Ferrari
Speed: 350 km/h
Turbo: Yes

F:\VJIT\JAVA\LAB\Unit-II\PART 1>_
```

### 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 classes B, C, and D.



*//Program on Hierarchical Inheritance*

*// Superclass: Vehicle*

class **Vehicle**

{

String brand;

int speed;

public **Vehicle**(String brand, int speed)

{

this.brand = brand;

this.speed = speed;

}

public void **displayInfo**()

{

System.out.println("Brand: " + brand);

System.out.println("Speed: " + speed + " km/h");

}

}

*// Subclass: Car*

class **Car extends Vehicle**

{

int numberOfDoors;

public **Car**(String brand, int speed, int numberOfDoors)

{

super(brand, speed);

this.numberOfDoors = numberOfDoors;

}

public void **displayCarInfo**()

{

displayInfo();

System.out.println("Number of Doors: " + numberOfDoors);

}

}



*// Subclass: Truck*

```
class Truck extends Vehicle
{
    int loadCapacity;
    public Truck(String brand, int speed, int loadCapacity)
    {
        super(brand, speed);
        this.loadCapacity = loadCapacity;
    }
    public void displayTruckInfo()
    {
        displayInfo();
        System.out.println("Load Capacity: " + loadCapacity + " tons");
    }
}
```

*//Main Class*

```
public class Hierarchical
{
    public static void main(String[] args)
    {
        Car c = new Car("Honda", 160, 4);
        Truck t = new Truck("Volvo", 120, 15);

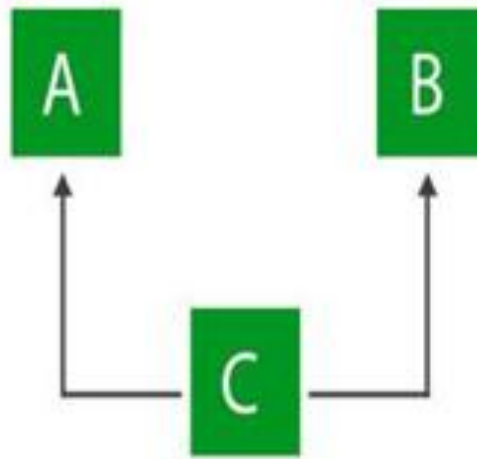
        c.displayCarInfo();
        t.displayTruckInfo();
    }
}
```

Command Prompt

```
F:\VJIT\JAVA\LAB\Unit-II\PART 1>javac Hierarchical.java
F:\VJIT\JAVA\LAB\Unit-II\PART 1>java Hierarchical
Brand: Honda
Speed: 160 km/h
Number of Doors: 4
Brand: Volvo
Speed: 120 km/h
Load Capacity: 15 tons
F:\VJIT\JAVA\LAB\Unit-II\PART 1>_
```

#### 4. Multiple Inheritance (Through Interfaces)

In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does **not** support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces. In the image below, Class C is derived from interfaces A and B.



Multiple Inheritance

*//Program to implement Multiple Inheritance*

*// Interface 1: Speed*

```
interface Speed
{
    void setSpeed(int speed);
    void displaySpeed();
}
```

*// Interface 2: Fuel*

```
interface Fuel
{
    void setFuelType(String fuelType);
    void displayFuelType();
}
```

*// Concrete class: Vehicle implements both interfaces*

class Vehicle **implements** Speed, Fuel

{

String brand;

int speed;

String fuelType;

public **Vehicle**(String brand)

{

    this.brand = brand;

}

@Override

public void **setSpeed**(int speed)

{

    this.speed = speed;

}

@Override

public void **displaySpeed**()

{

    System.out.println("Speed: " + speed + " km/h");

}

@Override

public void **setFuelType**(String fuelType)

{

    this.fuelType = fuelType;

}

@Override

public void **displayFuelType**()

{

    System.out.println("Fuel Type: " + fuelType);

}

```
public void displayBrand()
{
    System.out.println("Brand: " + brand);
}
}
//Main class
public class Multiple
{
    public static void main(String[] args)
    {
        Vehicle v = new Vehicle("Tesla");
        v.setSpeed(200);
        v.setFuelType("Electric");

        v.displayBrand();
        v.displaySpeed();
        v.displayFuelType();
    }
}
```

Command Prompt

```
F:\VJIT\JAVA\LAB\Unit-II\PART 1>javac Multiple.java

F:\VJIT\JAVA\LAB\Unit-II\PART 1>java Multiple
Brand: Tesla
Speed: 200 km/h
Fuel Type: Electric

F:\VJIT\JAVA\LAB\Unit-II\PART 1>
```

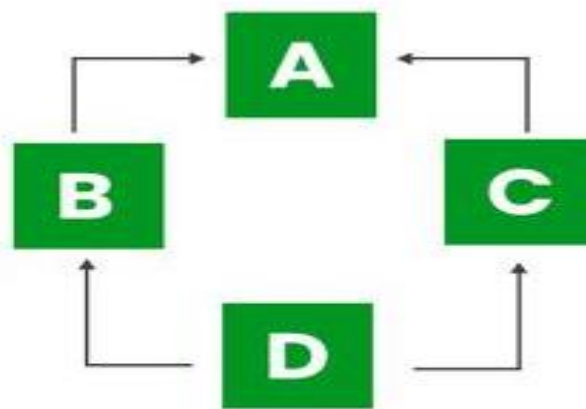
## 5. Hybrid Inheritance

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 involving multiple inheritance is also not possible with classes. In Java, we can achieve hybrid inheritance only through **Interfaces** if we want to involve multiple inheritance to implement Hybrid inheritance.

However, it is important to note that Hybrid inheritance does not necessarily require the use of Multiple Inheritance exclusively.

- It can be achieved through a combination of Multilevel Inheritance and Hierarchical Inheritance with classes.
- Hierarchical and Single Inheritance with classes.

Therefore, it is indeed possible to implement Hybrid inheritance using classes alone, without relying on multiple inheritance type.



**Hybrid Inheritance**

*//Program on Hybrid Inheritance*

*// Interface 1: Speed*

```
interface Speed
{
    void setSpeed(int speed);
    void displaySpeed();
}
```

*// Interface 2: Engine*

```
interface Engine
{
    void startEngine();
    void stopEngine();
}
```

```
// Superclass: Vehicle
class Vehicle
{
    String brand;

    public Vehicle(String brand)
    {
        this.brand = brand;
    }

    public void displayBrand()
    {
        System.out.println("Brand: " + brand);
    }
}

// Subclass: Car implements both interfaces (Speed, Engine)
class Car extends Vehicle implements Speed, Engine
{
    int speed;

    public Car(String brand)
    {
        super(brand);
    }

    @Override
    public void setSpeed(int speed)
    {
        this.speed = speed;
    }

    @Override
    public void displaySpeed()
    {
        System.out.println("Speed: " + speed + " km/h");
    }
}
```

```
@Override
public void startEngine()
{
    System.out.println("Engine started.");
}

@Override
public void stopEngine()
{
    System.out.println("Engine stopped.");
}
}

//Main class
public class Hybrid
{
    public static void main(String[] args)
    {
        Car c = new Car("BMW");
        c.displayBrand();
        c.setSpeed(220);
        c.displaySpeed();
        c.startEngine();
        c.stopEngine();
    }
}
```

Command Prompt

```
F:\VJIT\JAVA\LAB\Unit-II\PART 1>java Hybrid
Brand: BMW
Speed: 220 km/h
Engine started.
Engine stopped.

F:\VJIT\JAVA\LAB\Unit-II\PART 1>
```

## Forms of Inheritance in Java

The following are the different forms of inheritance in java.

1. Specialization
2. Specification
3. Construction
4. Extension or Generalization
5. Limitation
6. Combination

### 1. 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.

### 2. 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.

### 3. 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.

### 4. Extension or Generalization

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

### 5. Limitation

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

### 6. Combination

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



## Member access rules in Inheritance

### 1. *public Members*

- **Superclass:** public members of the superclass are accessible from any class, including subclasses, regardless of the package.
- **Subclass:** A subclass can access public members of its superclass without any restriction.

### 2. *private Members*

- **Superclass:** private members of the superclass **cannot** be accessed directly by any other class, including subclasses. These members are **hidden** from the subclass.
- **Subclass:** A subclass cannot access the private members of the superclass directly. However, the subclass can still access private members via **getter/setter methods** or other public/protected methods provided by the superclass.

### 3. *protected Members*

- **Superclass:** protected members of the superclass are accessible within the same package and in subclasses (even if they are in different packages).
- **Subclass:** A subclass can access protected members of its superclass, whether the subclass is in the same package or in a different package.

### 4. *default (Package-specific) Members*

- **Superclass:** Members with **default** access (no modifier) are accessible only within the same package.
- **Subclass:** If the subclass is in the same package, it can access these members. If the subclass is in a different package, it **cannot** access these members.

*// Vehicle class with different access modifiers*

class **Vehicle**

{

*//Can be accessed from anywhere*

**public** String brand;

*//Can only be accessed within the Vehicle class*

**private** int year;

*//Can be accessed within the package and by subclasses*

**protected** String model;

*//default Can be accessed only within the same package*

String color;

**public Vehicle**(String brand, int year, String model, String color)

{

    this.brand = brand;

    this.year = year;

    this.model = model;

    this.color = color;

}

*// Public method to access the private variable*

**public** int **getYear**()

{

    return year;

}

*// Only accessible within the Vehicle class*

**private** void **displayPrivateInfo**()

{

    System.out.println("Private method: Year of the vehicle is " + year);

}

*//Accessible within the package and by subclasses*

**protected** void **displayModel**()

{

    System.out.println("Model: " + model);

}

}

*// Subclass of Vehicle*

class Car **extends** Vehicle

{

    public **Car**(String brand, int year, String model, String color)

    {

        super(brand, year, model, color);

    }

*// Subclass can access protected member and method*

    public void **displayCarInfo**()

    {

        System.out.println("Brand: " + brand);

        System.out.println("Model (from subclass): " + model); *// access protected member*

        displayModel(); *// Can call protected method*

    }

}

*// Main class*

public class **Access**

{

    public static void **main**(String[] args)

    {

        Vehicle v = new Vehicle("Toyota", 2020, "Corolla", "Red");

*// Public member can be accessed directly*

        System.out.println("Brand: " + v.brand);

*// Cannot access private member directly, so using the public getter method*

        System.out.println("Year: " + v.getYear());

*// Cannot access private method directly*

*//v.displayPrivateInfo(); // This would cause a compile-time error*

*// Accessing default/package-private member within the same package*

        System.out.println("Color: " + v.color);

*// Subclass can access protected members and methods*

        Car c = new Car("Honda", 2021, "Civic", "Blue");

        c.displayCarInfo();

    }

}

```
F:\VJIT\JAVA\LAB\Unit-II\PART 1>javac Access.java

F:\VJIT\JAVA\LAB\Unit-II\PART 1>java Access
Brand: Toyota
Year: 2020
Color: Red
Brand: Honda
Model (from subclass): Civic
Model: Civic

F:\VJIT\JAVA\LAB\Unit-II\PART 1>
```

## super keyword

The super keyword in Java is used as a **reference variable** to **access objects** from **parent classes**. It allows access to parent class constructors, members, and methods in the derived class. The keyword plays a crucial role in inheritance and polymorphism concepts.

### Use of super keyword in Java

1. **super in Variables:** Used to access a variables from the superclass, especially if the subclass has a field with the same name.
2. **super in Methods:** Used to call a method from the superclass, especially when the method is overridden in the subclass.
3. **Super() in Constructors:** Used to invoke a constructor of the superclass. If the superclass doesn't have a no-argument constructor, you need to call one of its constructors explicitly using **super()**.

```
//Program on Super Keyword
// Superclass
class Vehicle
{
    String brand;
    int year;
    // Constructor of the superclass
    Vehicle(String brand, int year)
    {
        this.brand = brand;
        this.year = year;
    }
    // Method in the superclass
    void displayInfo()
    {
        System.out.println("Brand: " + brand);
    }
}
// Subclass
class Car extends Vehicle
{
    int doors;
    // Constructor of the subclass
    Car(String brand, int year, int doors)
    {
        // Using super to call the superclass constructor
        super(brand, year);
        this.doors = doors;
    }
    // Method in the subclass
    void displayCarInfo()
    {
        // Using super to call the superclass method
        super.displayInfo();
        System.out.println("Year: " + super.year);
        System.out.println("Doors: " + doors);
    }
}
```

```
// Main class
public class SuperKey
{
    public static void main(String[] args)
    {
        Car c = new Car("Toyota", 2020, 4);
        c.displayCarInfo();
    }
}
```

Command Prompt

```
F:\VJIT\JAVA\LAB\Unit-II\PART 1>javac SuperKey.java

F:\VJIT\JAVA\LAB\Unit-II\PART 1>java SuperKey
Brand: Toyota
Year: 2020
Doors: 4

F:\VJIT\JAVA\LAB\Unit-II\PART 1>_
```

### Restrictions on super:

- super can only be used to access members of the immediate superclass.
- super cannot be used to access private members of the superclass directly, as those are not visible outside the class.
- super cannot be used to call static methods or static variables of the superclass.

## Polymorphism

The term "polymorphism" means "many forms". In object-oriented programming, polymorphism is useful when you want to create multiple forms with the same name of a single entity.

Polymorphism in Java is mainly of **2 types** as mentioned below:

1. Method Overloading
2. Method Overriding

1. **Method Overloading:** Also, known as **compile-time** polymorphism, is the concept of Polymorphism where more than one method share the same name with different signature (Parameters) in a class. The return type of these methods can or cannot be same.
2. **Method Overriding:** Also, known as **run-time** polymorphism, is the concept of Polymorphism where method in the child class has the same name, return-type and parameters as in parent class. The child class provides the implementation in the method already written in the parent class.

*//Program on method overloading & overriding*

*// Parent Class*

**class Parent**

{

*// Method Defined*

**public void m1()**

{

System.out.println("Parent Method m1 executed");

}

*// Method Overloading*

**public void m1(int a)**

{

System.out.println("Parent Method m1 with parameter: " + a);

}

}

*// Child Class*

**class Child extends Parent**

{

*// Method Overriding*

**public void m1(int a)**

{

System.out.println("Child Method m1 with parameter: " + a);

}

}

*// Main Method*

```
class Poly
{
    public static void main(String args[])
    {
        Parent p = new Parent();
        p.m1();
        p.m1(5);
        Child c = new Child();
        c.m1(4);
    }
}
```

Command Prompt

```
F:\VJIT\JAVA\LAB>java Poly
Parent Method m1 executed
Parent Method m1 with parameter: 5
Child Method m1 with parameter: 4

F:\VJIT\JAVA\LAB>_
```

*//Program on method overriding*

*// Superclass vehicle*

```
class Vehicle
{
    String brand;
    public Vehicle(String brand)
    {
        this.brand = brand;
    }

    // Method that will be overridden by subclasses
    public void hornSound()
    {
        System.out.println(brand + " horn sound: Beep Beep!");
    }
}
```



```
// Subclass of Vehicle
class Car extends Vehicle
{
    public Car(String brand)
    {
        super(brand);
    }
    // Overriding the hornSound method in the Car class
    public void hornSound()
    {
        System.out.println(super.brand + " horn sound: Honk Honk!");
    }
}
//Main class
public class Override
{
    public static void main(String[] args)
    {
        Vehicle v = new Vehicle("Toyota");
        v.hornSound();
        Car c = new Car("Honda");
        c.hornSound();
    }
}
```

Command Prompt

```
F:\VJIT\JAVA\LAB\Unit-II\PART 1>javac Override.java

F:\VJIT\JAVA\LAB\Unit-II\PART 1>java Override
Toyota horn sound: Beep Beep!
Honda horn sound: Honk Honk!

F:\VJIT\JAVA\LAB\Unit-II\PART 1>_
```

## abstract classes

### Abstraction in Java

**Abstraction** is a process of showing the essential features and hiding their implementation details to the user.

**There are two ways to achieve abstraction in Java:**

1. Using Abstract Classes (0 to 100%)
2. Using Interfaces (100%)

### 1. Abstract Class

An abstract class in Java acts as a partially implemented class that itself cannot be instantiated. It exists only for subclassing purposes, and provides a template for its subcategories to follow. Abstract classes can have implementations with abstract methods. Abstract methods are declared to have no body, leaving their implementation to subclasses.

```
abstract class ClassName
{
    // Abstract method (no implementation)
    abstract void abstractMethod();

    // Concrete method (with implementation)
    void concreteMethod()
    {
        System.out.println("This is a concrete method");
    }
}
```

### Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- 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.
- We can't use final and abstract for a class at a time.

*//Program to implement abstract classes*

*// Abstract class: Vehicle*

**abstract class Vehicle**

```
{  
    String brand;  
    int speed;  
    // Constructor  
    public Vehicle(String brand, int speed)  
    {  
        this.brand = brand;  
        this.speed = speed;  
    }  
}
```

*// Abstract method: This method must be implemented by subclasses*

**abstract void startEngine();**

*// Regular method: Subclasses can inherit this directly*

```
public void displayInfo()  
{  
    System.out.println("Brand: " + brand);  
    System.out.println("Speed: " + speed + " km/h");  
}  
}
```

*// Subclass: Car*

**class Car extends Vehicle**

```
{  
    public Car(String brand, int speed)  
    {  
        super(brand, speed);  
    }  
    // Implementing the abstract method startEngine  
    void startEngine()  
    {  
        System.out.println("Car engine started.");  
    }  
}
```

```
// Subclass: Truck
class Truck extends Vehicle
{
    int loadCapacity; // in tons
    public Truck(String brand, int speed, int loadCapacity)
    {
        super(brand, speed);
        this.loadCapacity = loadCapacity;
    }
// Implementing the abstract method startEngine
    void startEngine()
    {
        System.out.println("Truck engine started.");
    }
// Method specific to Truck class
    public void displayLoadCapacity()
    {
        System.out.println("Load Capacity: " + loadCapacity + " tons");
    }
}
//Main class
public class Abstraction
{
    public static void main(String[] args)
    {
        // Creating objects of Car and Truck
        Car c = new Car("Toyota", 180);
        Truck t = new Truck("Volvo", 100, 10);
        // Calling methods
        c.startEngine();
        c.displayInfo();
        t.startEngine();
        t.displayInfo();
        t.displayLoadCapacity();
    }
}
```

Command Prompt

```
F:\VJIT\JAVA\LAB\Unit-II\PART 1>javac Abstraction.java

F:\VJIT\JAVA\LAB\Unit-II\PART 1>java Abstraction
Car engine started.
Brand: Toyota
Speed: 180 km/h
Truck engine started.
Brand: Volvo
Speed: 100 km/h
Load Capacity: 10 tons

F:\VJIT\JAVA\LAB\Unit-II\PART 1>_
```

## final keyword

The final keyword in Java is used to define constants, prevent method overriding, and prevent inheritance. It can be applied to variables, methods, and classes, each serving a different purpose.

### Uses of the final Keyword in Java:

1. **Final Variables (Constants):** When you declare a variable as final, its value cannot be changed once it is assigned. Essentially, the variable becomes a constant.
2. **Final Methods:** A final method cannot be overridden by subclasses. This is useful when you want to ensure that a particular method's behavior remains unchanged.
3. **Final Classes:** A final class cannot be subclassed (inherited). This is useful when you want to prevent a class from being extended.
4. **Final Parameters:** You can also use the final keyword on method parameters, which makes the parameters immutable within the method.

```
//program on final keyword
// Final class: Cannot be subclassed
final class Vehicle
{
    final String brand; //Cannot be changed once assigned
    final int year;    //Cannot be changed once assigned

    // Constructor
    Vehicle(String brand, int year)
    {
        this.brand = brand;
        this.year = year;
    }
    // Method with final parameter
    public void setVehicleType(final String vehicleType)
    {
        // Uncommenting the following line would cause a compilation error
        // vehicleType = "Truck"; // Error: cannot assign a value to final variable
        System.out.println("Vehicle type is: " + vehicleType);
    }
    // Final method: Cannot be overridden by subclasses
    final void displayInfo()
    {
        System.out.println("Brand: " + brand + ", Year: " + year);
    }
}

// Uncommenting the following line will cause a compile-time error because Vehicle
is final
// class Car extends Vehicle
// {

// }
```

```
// Main class
public class FinalKey
{
    public static void main(String[] args)
    {
        Vehicle v = new Vehicle("Toyota", 2020);
        // Passing a String parameter to the setVehicleType method
        v.setVehicleType("Car");
        // The final variable brand cannot be reassigned
        // v.brand = "Honda"; // This would cause a compile-time error
        v.displayInfo(); // Calling the final method
    }
}
```

Command Prompt

```
F:\VJIT\JAVA\LAB\Unit-II\PART 1>javac FinalKey.java

F:\VJIT\JAVA\LAB\Unit-II\PART 1>java FinalKey
Vehicle type is: Car
Brand: Toyota, Year: 2020

F:\VJIT\JAVA\LAB\Unit-II\PART 1>
```

## Interfaces and Packages

**Interfaces:** *Introduction to Interfaces, differences between abstract classes and interfaces, multiple inheritance through interfaces,*

**Packages:** *Creating and accessing a package, Understanding CLASSPATH, importing packages.*

## Interfaces

Interfaces in Java are a collection of abstract and public methods we want our classes to implement. It is the blueprint of a class and contains static constants and abstract methods. Interfaces are fully unimplemented one.

Interfaces are used to achieve abstraction and implement multiple inheritance.

### Syntax of Interface

```
interface InterfaceName
{
    void method1(); // Abstract method (no body)
    // Default method with implementation
    default void method2()
    {
        System.out.println("This is a default method in the interface.");
    }
    // Static method with implementation
    static void method3()
    {
        System.out.println("This is a static method in the interface.");
    }
}
```

### Key Features of Interfaces:

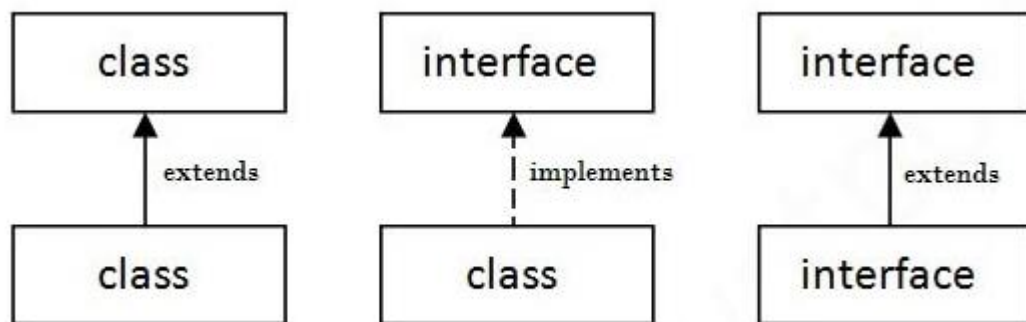
1. **Abstract Methods:** All methods in an interface are implicitly abstract (unless they are default or static methods).
2. **No Method Body:** An interface only defines method signatures; it does not provide method implementations (except for default and static methods).
3. **Multiple Inheritance:** A class can implement multiple interfaces, which allows for multiple inheritance of behavior.



4. **Implementing Interfaces:** A class that implements an interface must provide concrete implementations for all of the interface's abstract methods (unless the class is abstract).
5. **default and static Methods:** Since Java 8, interfaces can contain default methods with an implementation, as well as static methods.

### Relationship Between Classes and Interfaces

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



### Important Points about interfaces

- We can't create an instance (interface can't be instantiated) of the interface but we can make the reference of it that refers to the Object of its implementing class.
- A class can implement more than one interface.
- An interface can extend to another interface or interface (more than one interface).
- A class that implements the interface must implement all the methods in the interface.
- All the methods are public and abstract. All the fields are public, static, and final.
- It is used to achieve multiple inheritances.
- Inside the Interface not possible to declare instance variables because by default variables are **public static final**.
- Inside the Interface, constructors are not allowed.
- Inside the interface main method is not allowed.
- Inside the interface, static, final, and private methods declaration are not possible.

*//Program to implement interfaces*

*// Interface: Vehicle*

**interface** Vehicle

```
{  
    // Abstract methods (no body)  
    void start();  
    void stop();  
    void accelerate();  
    void honk();  
}
```

*// Class Car implementing the Vehicle interface*

class Car **implements** Vehicle

```
{  
    private String brand;  
    private String model;  
    // Constructor  
    public Car(String brand, String model)  
    {  
        this.brand = brand;  
        this.model = model;  
    }  
    // Implementing the start method from the Vehicle interface  
    public void start()  
    {  
        System.out.println(brand + " " + model + " is starting.");  
    }  
    // Implementing the stop method from the Vehicle interface  
    public void stop()  
    {  
        System.out.println(brand + " " + model + " is stopping.");  
    }  
    // Implementing the accelerate method from the Vehicle interface  
    public void accelerate()  
    {  
        System.out.println(brand + " " + model + " is accelerating.");  
    }  
}
```

```
// Implementing the honk method from the Vehicle interface
public void honk()
{
    System.out.println(brand + " " + model + " is honking: Beep Beep!");
}
}

// Class Truck implementing the Vehicle interface
class Truck implements Vehicle
{
    private String brand;
    private String model;
    public Truck(String brand, String model)
    {
        this.brand = brand;
        this.model = model;
    }

    // Implementing the start method from the Vehicle interface
    public void start()
    {
        System.out.println(brand + " " + model + " truck is starting.");
    }

    // Implementing the stop method from the Vehicle interface
    public void stop()
    {
        System.out.println(brand + " " + model + " truck is stopping.");
    }

    // Implementing the accelerate method from the Vehicle interface
    public void accelerate()
    {
        System.out.println(brand + " " + model + " truck is accelerating.");
    }

    // Implementing the honk method from the Vehicle interface
    public void honk()
    {
        System.out.println(brand + " " + model + " truck is honking: Horn Horn!");
    }
}
```

*// Main class to demonstrate the usage of Vehicle interface*

```
public class InterfaceEx
```

```
{
```

```
    public static void main(String[] args)
```

```
    {
```

```
        // Create objects of Car and Truck
```

```
        Vehicle c = new Car("Toyota", "Camry");
```

```
        Vehicle t = new Truck("Ford", "F-150");
```

```
        // Demonstrate using the interface methods
```

```
        System.out.println("Car Actions:");
```

```
        c.start();
```

```
        c.accelerate();
```

```
        c.honk();
```

```
        c.stop();
```

```
        System.out.println("\nTruck Actions:");
```

```
        t.start();
```

```
        t.accelerate();
```

```
        t.honk();
```

```
        t.stop();
```

```
    }
```

```
}
```

Command Prompt

```
F:\VJIT\JAVA\LAB\Unit-II\PART 2>javac InterfaceEx.java
```

```
F:\VJIT\JAVA\LAB\Unit-II\PART 2>java InterfaceEx
```

```
Car Actions:
```

```
Toyota Camry is starting.
```

```
Toyota Camry is accelerating.
```

```
Toyota Camry is honking: Beep Beep!
```

```
Toyota Camry is stopping.
```

```
Truck Actions:
```

```
Ford F-150 truck is starting.
```

```
Ford F-150 truck is accelerating.
```

```
Ford F-150 truck is honking: Horn Horn!
```

```
Ford F-150 truck is stopping.
```

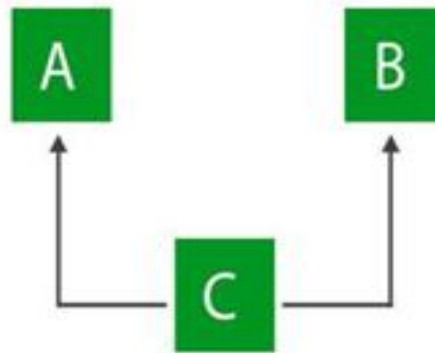
```
F:\VJIT\JAVA\LAB\Unit-II\PART 2>_
```

## Differences between abstract classes and interfaces

S.No	Abstract class	Interface
1.	The <b>abstract</b> keyword is used to declare abstract class.	The <b>interface</b> keyword is used to declare interface.
2.	It has partially implemented and partially unimplemented methods	It has fully unimplemented methods
3.	Abstract class can <b>have abstract and non-abstract</b> methods.	Interface can have <b>only abstract</b> methods. Since Java 8, it can have <b>default and static methods</b> also.
4.	An abstract class can be extended using keyword " <b>extends</b> ".	An <b>interface</b> can be implemented using keyword " <b>implements</b> ".
5.	An abstract class can have class members like <b>private, protected</b> , etc.	Members of interface are <b>public</b> by default.
6.	Abstract class can have <b>final, non-final, static and non-static variables</b> .	Interface has <b>only static and final variables</b> .
7.	Abstract Class can <b>have constructors</b>	Interface <b>cannot have constructors</b>
8.	Abstract class <b>can provide</b> the implementation of interface.	Interface <b>can't provide</b> the implementation of abstract class.
9.	An <b>abstract class</b> can extend another Java class and implement multiple Java interfaces.	An <b>interface</b> can extend another Java interface or multiple interfaces only.
10.	Abstract class <b>doesn't support multiple inheritance</b> .	Interface <b>supports multiple inheritance</b> .
11.	<b>Example:</b> public <b>abstract</b> class Shape { public <b>abstract</b> void draw(); }	<b>Example:</b> public <b>interface</b> Drawable { void draw(); }

### Multiple Inheritance through interfaces

In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does **not** support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces. In the image below, Class C is derived from interfaces A and B.



Multiple Inheritance

*//Program to implement Multiple Inheritance through interfaces*

*// Interface for Engine-related behavior*

**interface** Engine

```
{  
    void startEngine(); // Abstract method to start the engine  
    void stopEngine(); // Abstract method to stop the engine  
}
```

*// Interface for general vehicle behavior*

**interface** Vehicle

```
{  
    void drive(); // Abstract method for driving the vehicle  
    void honk(); // Abstract method for honking the vehicle  
}
```

```
// Car class that implements both Engine and Vehicle interfaces
class Car implements Engine, Vehicle
{
    private String brand;
    private String model;

    // Constructor to initialize the car's brand and model
    public Car(String brand, String model)
    {
        this.brand = brand;
        this.model = model;
    }

    // Implementing startEngine from Engine interface
    public void startEngine()
    {
        System.out.println(brand + " " + model + " engine started.");
    }

    // Implementing stopEngine from Engine interface
    public void stopEngine()
    {
        System.out.println(brand + " " + model + " engine stopped.");
    }

    // Implementing drive from Vehicle interface
    public void drive()
    {
        System.out.println(brand + " " + model + " is driving.");
    }

    // Implementing honk from Vehicle interface
    public void honk()
    {
        System.out.println(brand + " " + model + " is honking: Beep Beep!");
    }
}
```

```
// Main class to demonstrate multiple inheritance through interfaces
public class Multiple
{
    public static void main(String[] args)
    {
// Create an object of Car which implements both Engine and Vehicle interfaces
        Car c = new Car("Toyota", "Camry");
// Use the methods from both Engine and Vehicle interfaces
        c.startEngine(); // Engine-related behavior
        c.drive();       // Vehicle-related behavior
        c.honk();        // Vehicle-related behavior
        c.stopEngine(); // Engine-related behavior
    }
}
```

Command Prompt

```
F:\VJIT\JAVA\LAB\Unit-II\PART 2>javac Multiple.java

F:\VJIT\JAVA\LAB\Unit-II\PART 2>java Multiple
Toyota Camry engine started.
Toyota Camry is driving.
Toyota Camry is honking: Beep Beep!
Toyota Camry engine stopped.

F:\VJIT\JAVA\LAB\Unit-II\PART 2>
```

## Packages

### Creating and accessing a package

In Java, packages are used to group related classes and interfaces together. This helps in organizing code in a logical manner and avoiding name conflicts. A package also controls access to classes and members, helping with encapsulation.

#### Types of Packages:

1. **Built-in Packages:** These are provided by the Java API (e.g., java.util, java.io, java.lang). They come pre-installed with the Java Development Kit (JDK).
2. **User-defined Packages:** These are packages that you define for your own classes and interfaces.



### Importance of Packages?

1. **Organization:** Packages allow you to logically group related classes and interfaces, making it easier to maintain and manage code.
2. **Name Conflict Avoidance:** By placing classes in different packages, you can avoid name conflicts between classes with the same name.
3. **Access Control:** Packages provide a way to control access to classes, methods, and variables using access modifiers like private, protected, and public.
4. **Reusability:** Packages allow code to be reused across different projects by creating modular and organized libraries.

### Syntax for Declaring a Package

At the top of a Java source file, you can declare the package it belongs to using the package keyword:

```
package package_name;
```

Ex:

```
package com.mahindra.utility;  
  
public class Vehicle  
{  
    // class implementation  
}
```

### 1. Built-In Packages

In Java, **built-in packages** are predefined and included in the Java Standard Library (Java API). These packages provide a wide range of functionality that can be used directly in your programs without having to write the code yourself.

The Java standard library contains packages that allow you to work with data structures, file handling, networking, utilities, and more.

Here are some of the most commonly used **built-in packages** in Java:

### 1. java.lang Package

- **Description:** This is one of the most fundamental packages in Java. It is automatically imported in every Java program, so you don't need to explicitly import it.
- **Classes and Interfaces:**
  - String: Represents strings of characters.
  - Math: Provides mathematical functions like `sqrt()`, `pow()`, etc.
  - Object: The root class of the Java class hierarchy.
  - System: Provides system-related utility methods (e.g., `System.out.println()` ).
  - Thread: Represents a thread of execution.
  - Exception, RuntimeException: Represents exceptions in Java.

### 2. java.util Package

- **Description:** Provides classes and interfaces for data structures, date and time, and utility classes.
- **Classes and Interfaces:**
  - ArrayList: A resizable array implementation of the List interface.
  - HashMap: Implements the Map interface, storing key-value pairs.
  - Date: Represents date and time.
  - Collections: Contains static methods for manipulating collections (e.g., sorting, reversing).
  - Scanner: Used to get input from the user.

### 3. java.io Package

- **Description:** Contains classes and interfaces for input and output (I/O) operations like reading from or writing to files, and working with data streams.
- **Classes and Interfaces:**
  - File: Represents a file or directory path.
  - FileReader, BufferedReader: Used for reading data from files.
  - FileWriter, BufferedWriter: Used for writing data to files.
  - InputStream, OutputStream: For byte-based I/O.
  - ObjectInputStream, ObjectOutputStream: For object serialization.

#### 4. java.net Package

- **Description:** Provides classes for networking operations such as sending and receiving data over the network.
- **Classes and Interfaces:**
  - Socket: Used for creating client-side TCP connections.
  - ServerSocket: Used for creating server-side TCP connections.
  - URL: Represents a Uniform Resource Locator (URL).
  - URLConnection: Provides methods for communicating with URLs.

#### 5. java.sql Package

- **Description:** Provides classes and interfaces for working with databases using JDBC (Java Database Connectivity).
- **Classes and Interfaces:**
  - Connection: Represents a connection to a database.
  - Statement: Used to execute SQL queries.
  - ResultSet: Represents the result set of a query.
  - DriverManager: Manages database drivers.

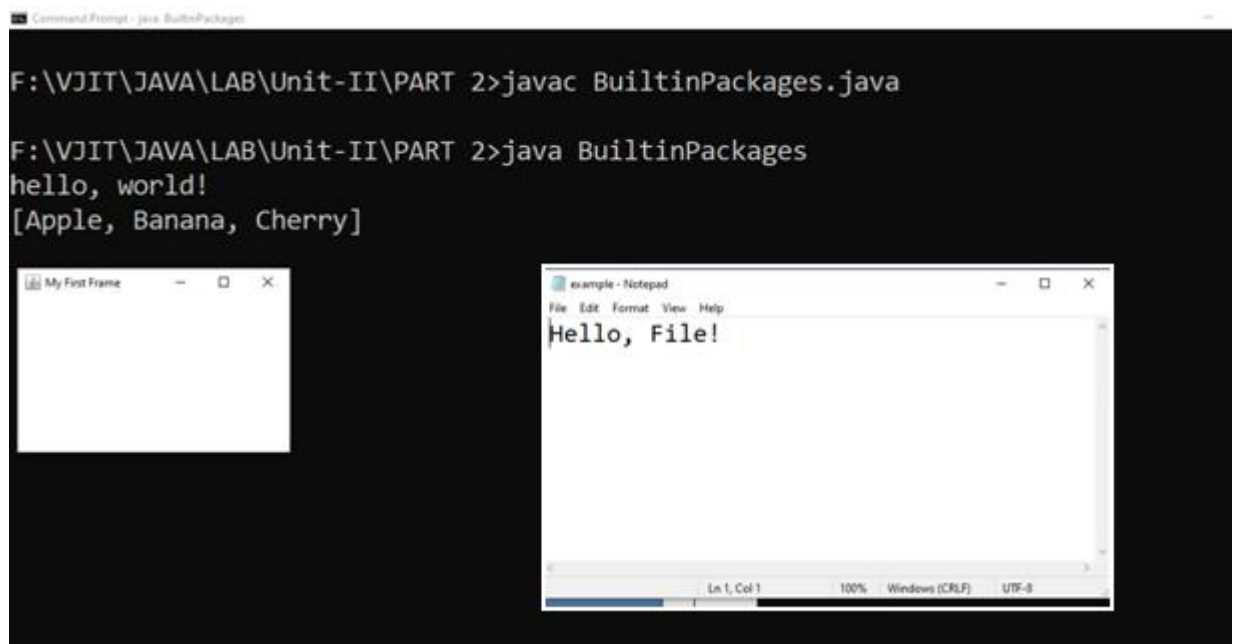
#### 6. java.awt Package

- **Description:** The java.awt (Abstract Window Toolkit) package is part of Java's **Java Foundation Classes (JFC)** and provides a set of APIs for building Graphical User Interfaces (GUIs) in Java.
- It contains classes for building windows, buttons, text fields, layouts, and other graphical components that can be used in Java desktop applications.

*//Program to implement Built-in Packages*

```
import java.lang.*;
import java.util.ArrayList;
import java.io.*;
import java.io.IOException;
import java.awt.*;

public class BuiltinPackages
{
    public static void main(String[] args) throws IOException
    {
        //lang package
        String text = "Hello, World!";
        System.out.println(text.toLowerCase());
        //util package
        ArrayList<String> list = new ArrayList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Cherry");
        System.out.println(list);
        //io package
        File f = new File("example.txt");
        FileWriter fw = new FileWriter(f);
        fw.write("Hello, File!");
        fw.close();
        //awt package
        Frame fr = new Frame("My First Frame");
        fr.setSize(300, 200);
        fr.setVisible(true);
    }
}
```



The screenshot shows a Windows environment with two windows. The background is a black Command Prompt window titled 'Command Prompt - java BuiltinPackages'. It displays the following commands and output:  
F:\VJIT\JAVA\LAB\Unit-II\PART 2>javac BuiltinPackages.java  
F:\VJIT\JAVA\LAB\Unit-II\PART 2>java BuiltinPackages  
hello, world!  
[Apple, Banana, Cherry]  
In the foreground, there are two smaller windows. On the left is a window titled 'My First Frame' which is currently empty. On the right is a Notepad window titled 'example - Notepad' containing the text 'Hello, File!'.

## 2. User Defined Packages

**In Java, user-defined packages** allow you to organize your classes, interfaces, and sub-packages into separate namespaces for better management and structure.

By grouping related classes into packages, you can avoid naming conflicts and improve code readability and maintainability.

### How to Create and Use User-Defined Packages

*Steps to Create a User-Defined Package:*

1. **Create a Package:** To define a user-defined package, use the package keyword at the top of your Java source file.
2. **Compile the Class:** After defining the package, compile the Java file from the directory above the package directory.
3. **Import the Package:** If you want to use classes from a package in another class, use the import keyword.

Let's go through an example where we create a vehicle package, define classes related to vehicles in that package, and then access those classes from another package.

**Steps:**

1. Create a vehicle package with classes like Vehicle and Car.
2. Create a Main class in a different package to access and use those classes.

**Step 1: Creating the vehicle Package**

First, we create a package called vehicle and add two classes: Vehicle and Car.

**File: Vehicle.java (Inside vehicle package)**

**package** vehicle;

// Vehicle class with basic properties and methods

public class Vehicle

{

    private String brand;

    private int year;

    // Constructor

    public **Vehicle**(String brand, int year)

    {

        this.brand = brand;

        this.year = year;

    }

    // Getter and setter methods

    public String **getBrand**()

    {

        return brand;

    }

    public void **setBrand**(String brand)

    {

        this.brand = brand;

    }

    public int **getYear**()

    {

        return year;

    }

```
    public void setYear(int year)
    {
        this.year = year;
    }
    // Method to display basic vehicle information
    public void displayInfo()
    {
        System.out.println("Brand: " + brand + ", Year: " + year);
    }
}
```

**File: Car.java (Inside vehicle package)**

**package** vehicle;

// Car class extends Vehicle and adds more functionality

**public class** Car **extends** Vehicle

```
{
    private int doors;
    // Constructor
    public Car(String brand, int year, int doors)
    {
        super(brand, year); // Calling the superclass (Vehicle) constructor
        this.doors = doors;
    }
    // Getter and setter methods
    public int getDoors()
    {
        return doors;
    }
    public void setDoors(int doors)
    {
        this.doors = doors;
    }
    // Overridden method to display information about the car
    @Override
    public void displayInfo()
    {
        super.displayInfo(); // Call the displayInfo of Vehicle
        System.out.println("Doors: " + doors);
    }
}
```

**Step 2: Creating the Main Class to Access the vehicle Package**

Now, in a different package, we create a Main class to demonstrate how to use the Vehicle and Car classes.

**File: Main.java (In the default package or another package)**

```
//Main.java
import vehicle.Vehicle;
import vehicle.Car;

public class Main
{
    public static void main(String[] args)
    {
        // Create a Vehicle object
        Vehicle v = new Vehicle("Toyota", 2020);
        v.displayInfo();

        // Create a Car object (which is a subclass of Vehicle)
        Car c = new Car("Honda", 2022, 4);
        c.displayInfo(); // It will also display Vehicle's info
    }
}
```

**Explanation:****1. vehicle Package:**

- The Vehicle class has basic properties such as brand and year, along with methods to get and set those properties. It also has a displayInfo() method to print the vehicle's information.
- The Car class extends Vehicle and adds a doors property, representing the number of doors in the car. It overrides the displayInfo() method to include the number of doors in the output.

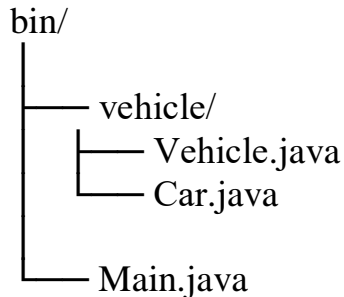
**2. Main Class:**

- The Main class imports the Vehicle and Car classes from the vehicle package using the import statement.
- It then creates instances of both the Vehicle and Car classes and calls their displayInfo() methods.



### Directory Structure:

To compile and run the above code, your directory structure should look like this:



### How to Compile and Run:

#### 1. Compile the classes & Run Main Class:

```
Command Prompt

F:\VJIT\JAVA\LAB\Unit-II\PART 2\User Defined packages>javac -d bin *.java

F:\VJIT\JAVA\LAB\Unit-II\PART 2\User Defined packages>cd bin

F:\VJIT\JAVA\LAB\Unit-II\PART 2\User Defined packages\bin>java Main
Brand: Toyota, Year: 2020
Brand: Honda, Year: 2022
Doors: 4

F:\VJIT\JAVA\LAB\Unit-II\PART 2\User Defined packages\bin>_
```

## Understanding CLASSPATH

In Java, the CLASSPATH is an environment variable that tells the Java Virtual Machine (JVM) and Java compiler where to look for compiled Java class files and other resources like libraries (JAR files) when executing a program.

When you run a Java program, the JVM needs to find the classes that are part of your program. By default, the JVM looks for classes in the current directory. However, if your classes are located in different directories or JAR files, you need to specify these locations via the CLASSPATH environment variable.

### What it can contain:

- **Directories:** These contain compiled .class files.
- **JAR files:** Java Archive files, which package multiple .class files and associated resources into a single file.

## Creating a JAR file

Whenever a developer wants to distribute a version of his software, then all he want is to distribute a single file and not a directory structure filled with class files. JAR files were designed for this purpose. A JAR file can contain both class files and other file types like sound and image files which may be included in the project. All the files in a JAR file are compressed using a format similar to zip.

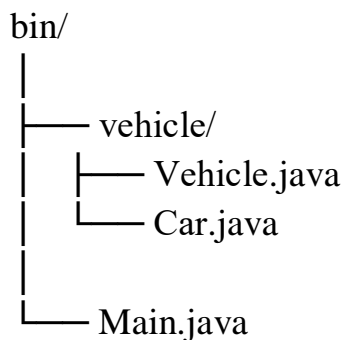
A jar file is created using jar tool. The general command looks somewhat like this:

```
jar options jar-file [manifest-file] file1 file2 file3 ...
```

- **jar – file** : name of jar file on which you want to use jar tool.
- **file1, file2, file3** : files which you want to add inside a jar file. manifest-file is the name of file which contains manifest of that jar file, giving manifest-file as an argument is entirely optional.
- **c** : Creates a new or empty archive and adds files to it. If any of the specified file name are directories, then the jar program processes them recursively.
- **C** : Temporarily changes the directory.
- **e** : Creates an entry point in the manifest.
- **f** : Specifies the JAR file name as the second command-line argument. If this parameter is missing, jar will write the result to standard output (when creating a JAR file) or read it from standard input (when extracting or tabulating a JAR file).
- **i** : Creates an index file.
- **m** : Adds a manifest file to the JAR file. A manifest is a description of the archive contents and origin. Every archive has a default manifest, but you can supply your own if you want to authenticate the contents of the archive.
- **M** : Does not create a manifest file for the entries.
- **t** : Displays the table of contents.
- **u** : Updates an existing JAR file.
- **v** : Generates verbose output.
- **x** : Extract files. If you supply one or more file names, only those files are extracted. Otherwise, all files are extracted.

### 1. Directory Structure:

To compile and run the above code, your directory structure should look like this:



### 2. Compile the Java Files

To compile the Java files, navigate to the src directory and compile the .java files:

```
Command Prompt
F:\VJIT\JAVA\LAB\Unit-II\PART 2\User Defined packages>javac -d bin *.java
F:\VJIT\JAVA\LAB\Unit-II\PART 2\User Defined packages>
```

This will compile the Java files and place the resulting .class files in the bin directory, following the package structure.

### 3. Create the Manifest File

To specify the entry point of your application (the class with the main method), you need to create a manifest file with the Main-Class attribute. The manifest file will tell the JVM which class to run when you execute the JAR file.

**Create a directory structure like this:**

```
/MyProject
/META-INF
  MANIFEST.MF
```

In the MANIFEST.MF file, add the following:

```
Manifest-Version: 1.0
Main-Class: Main
```

- **Main-Class:** The fully qualified class name of the class that contains the main method to execute. In this case, Main.

#### 4. Create the JAR File

You can now create the JAR file using the jar command. Use the -m option to specify your custom MANIFEST.MF file and the -C option to include your compiled .class files from the bin directory.

Run the following command:

```
jar cmf META-INF/MANIFEST.MF myapp.jar -C bin/ .
```

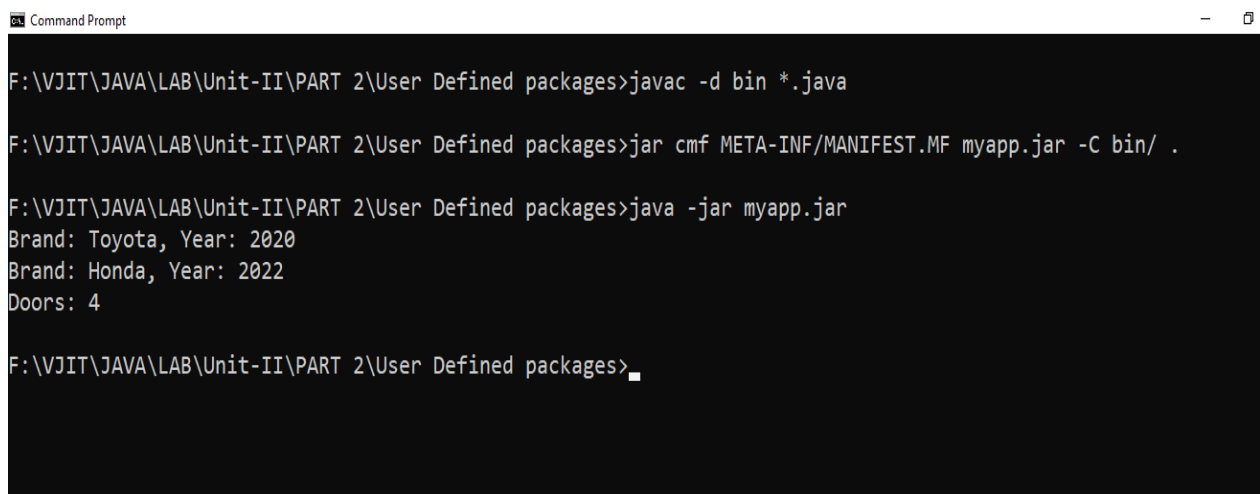
- c: Create a new JAR file.
- m: Specify the manifest file.
- f: Specify the JAR file name (myapp.jar).
- -C bin/: Include the compiled .class files from the bin directory.

After running the command, the myapp.jar file will be created, containing the classes and manifest.

#### 5. Run the JAR File

Once the JAR file is created, you can run it using the java -jar command. The java -jar command will automatically use the Main-Class attribute from the manifest file to determine which class to execute.

**To run the JAR file:**



```
Command Prompt
F:\VJIT\JAVA\LAB\Unit-II\PART 2\User Defined packages>javac -d bin *.java
F:\VJIT\JAVA\LAB\Unit-II\PART 2\User Defined packages>jar cmf META-INF/MANIFEST.MF myapp.jar -C bin/ .
F:\VJIT\JAVA\LAB\Unit-II\PART 2\User Defined packages>java -jar myapp.jar
Brand: Toyota, Year: 2020
Brand: Honda, Year: 2022
Doors: 4
F:\VJIT\JAVA\LAB\Unit-II\PART 2\User Defined packages>
```

**How to set CLASSPATH:**

You can set the CLASSPATH

- Temporarily or
- Permanently

***Temporarily (in a command line):***

To set the CLASSPATH temporarily for a session (only for the current command prompt), you can use the following:

```
set CLASSPATH=C:\path\to\classes;C:\path\to\jarfile.jar
```

***Permanently (in System Variable at as Environment variables):***

Set the CLASSPATH variable:

```
C:\path\to\your\bin;C:\path\to\libs\library1.jar;C:\path\to\libs\library2.jar
```

**Exception Handling****PART-3**

*Concepts of exception handling, exception hierarchy, built in exceptions, usage of try, catch, finally, throw, and throws, creating own exception sub classes.*

**Concepts of Exception Handling**

An **Exception** is an unwanted or unexpected event that occurs during the execution of a program (i.e., at **runtime**) and disrupts the normal flow of the program's instructions.

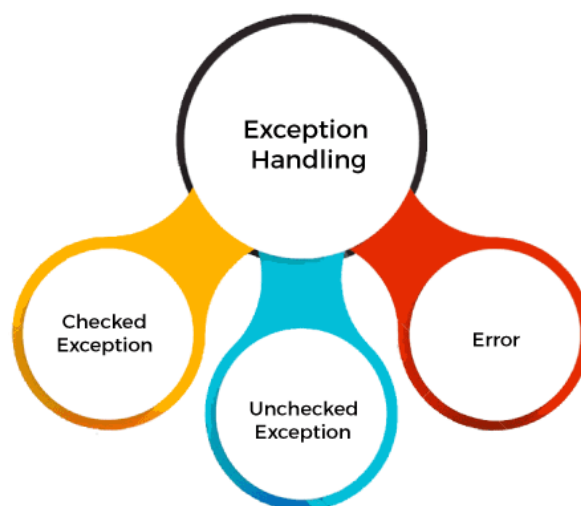
It occurs when something unexpected things happen, like accessing an invalid index, dividing by zero, or trying to open a file that does not exist.

**Exception handling in Java** allows developers to manage runtime errors effectively by using mechanisms like **try-catch block**, **finally block**, **throwing Exceptions**, **Custom Exception handling**, etc.

**Types of Java Exceptions**

In Java, exceptions are categorized into two main types: checked exceptions and unchecked exceptions. Additionally, there is a third category known as errors. Let's delve into each of these types:

1. Checked Exception
2. Unchecked Exception
3. Error



## 1. Checked Exceptions (Compile -time)

Checked exceptions are called **compile-time** exceptions because these exceptions are checked at compile-time by the compiler.

*Examples of Checked Exception are listed below:*

1. **ClassNotFoundException:** Throws when the program tries to load a class at runtime but the class is not found because its not present in the correct location or it is missing from the project.
2. **InterruptedException:** Thrown when a thread is paused and another thread interrupts it.
3. **IOException:** Throws when input/output operation fails
4. **InstantiationException:** Thrown when the program tries to create an object of a class but fails because the class is abstract, an interface, or has no default constructor.
5. **SQLException:** Throws when there's an error with the database.
6. **FileNotFoundException:** Thrown when the program tries to open a file that doesn't exist

## 2. Unchecked Exceptions (Runtime Exceptions)

The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time.

In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.

*Examples of Unchecked Exception are listed below:*

1. **ArithmeticException:** It is thrown when there's an illegal math operation.
2. **ClassCastException:** It is thrown when you try to cast an object to a class it does not belongs to.

3. **NullPointerException:** It is thrown when you try to use a null object (e.g. accessing its methods or fields)
4. **ArrayIndexOutOfBoundsException:** It occurs when we try to access an array element with an invalid index.
5. **ArrayStoreException:** It happens when you store an object of the wrong type in an array.
6. **IllegalThreadStateException:** It is thrown when a thread operation is not allowed in its current state

### 3. Errors

Errors represent exceptional conditions that are not expected to be caught under normal circumstances.

They are typically caused by issues outside the control of the application, such as system failures or resource exhaustion. Errors are not meant to be caught or handled by application code.

*Examples of errors include:*

1. **OutOfMemoryError:** It occurs when the Java Virtual Machine (JVM) cannot allocate enough memory for the application.
2. **StackOverflowError:** It is thrown when the stack memory is exhausted due to excessive recursion.
3. **NoClassDefFoundError:** It indicates that the JVM cannot find the definition of a class that was available at compile-time.

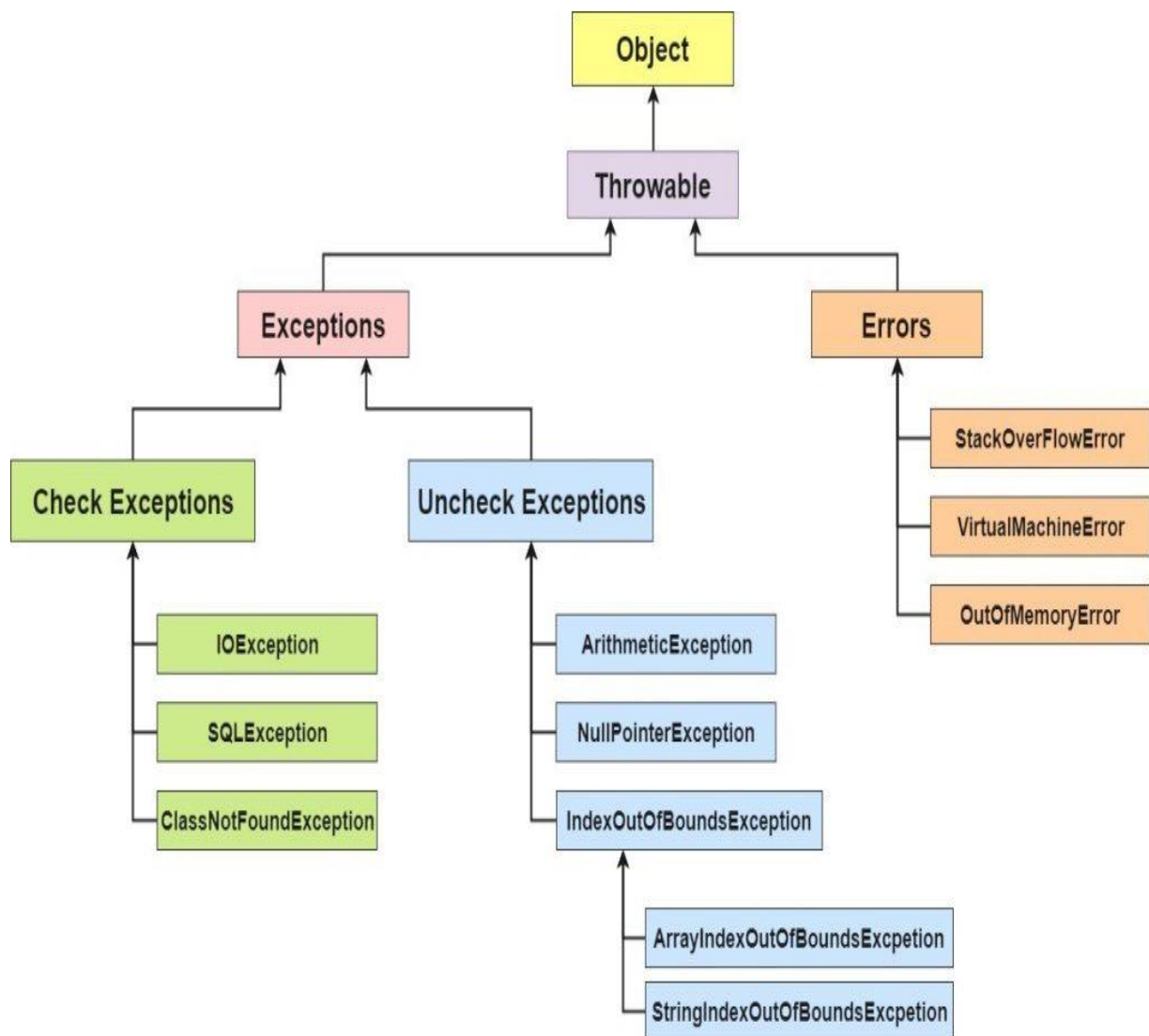


## Hierarchy of Exceptions in Java

All exception and error types are subclasses of the class **Throwable**, which is the base class of the hierarchy.

One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. `NullPointerException` is an example of such an exception.

Another branch, **Error** is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself (JRE). `StackOverflowError` is an example of such an error.



## Built-in Exceptions in Java

Built-in exceptions, also known as standard exceptions, are predefined exception classes provided by Java.

These exceptions cover a wide range of common errors and exceptional situations that can occur during program execution.

Built-in exceptions are part of the Java standard library and provide a standardized way to handle common exceptional scenarios.

*Built-In Exceptions can be further classified into two categories –*

1. Checked Exceptions
2. Unchecked Exceptions

### 1. Checked Exceptions(Compiletime Exceptions)

Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler. Examples of Checked Exception are listed below:

1. **ClassNotFoundException:** Throws when the program tries to load a class at runtime but the class is not found because its not present in the correct location or it is missing from the project.
2. **InterruptedException:** Thrown when a thread is paused and another thread interrupts it.
3. **IOException:** Throws when input/output operation fails
4. **InstantiationException:** Thrown when the program tries to create an object of a class but fails because the class is abstract, an interface, or has no default constructor.
5. **SQLException:** Throws when there's an error with the database.
6. **FileNotFoundException:** Thrown when the program tries to open a file that doesn't exist.

## 2. Unchecked Exceptions (Runtime Exceptions)

The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error. Examples of Unchecked Exception are listed below:

1. **ArithmeticException:** It is thrown when there's an illegal math operation.
2. **ClassCastException:** It is thrown when you try to cast an object to a class it does not belongs to.
3. **NullPointerException:** It is thrown when you try to use a null object (e.g. accessing its methods or fields)
4. **ArrayIndexOutOfBoundsException:** It occurs when we try to access an array element with an invalid index.
5. **ArrayStoreException:** It happens when you store an object of the wrong type in an array.
6. **IllegalThreadStateException:** It is thrown when a thread operation is not allowed in its current state

### Java Exception Class Methods

Following is the list of important methods available in the Throwable class.

S.No.	Method & Description
1.	<b>public String getMessage()</b> Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2.	<b>public String toString()</b> Returns the name of the class concatenated with the result of getMessage().
3.	<b>public void printStackTrace()</b> Prints the result of toString() along with the stack trace to System.err, the error output stream.

Let's go through an example where we use built-in exceptions, such as `IllegalArgumentException`, `ArithmeticException`, and `NullPointerException`, in the context of a `Vehicle` class.

### Example: Vehicle Built-in Exceptions in Java

*//Vehicle Built-in Exceptions in Java*

**package** vehicle;

**public class** **Vehicle**

```
{
    private String brand;
    private int fuelLevel; // Fuel level in percentage (0 to 100)
    private boolean isRunning;
    // Constructor to initialize vehicle brand and fuel level
    public Vehicle(String brand, int fuelLevel)
    {
        if (fuelLevel < 0 || fuelLevel > 100)
        {
            throw new IllegalArgumentException("Fuel level must be between 0 to 100.");
        }
        this.brand = brand;
        this.fuelLevel = fuelLevel;
        this.isRunning = false; // Initially the vehicle is not running
    }
    // Method to start the vehicle engine
    public void startEngine()
    {
        if (fuelLevel <= 0)
        {
            throw new ArithmeticException("Cannot start the engine. Fuel level is zero.");
        }
        if (isRunning)
        {
            System.out.println(brand + " engine is already running.");
        }
    }
}
```

```
        else
        {
            isRunning = true;
            System.out.println(brand + " engine started.");
        }
    }
    // Method to stop the vehicle engine
    public void stopEngine()
    {
        if (!isRunning)
        {
            throw new IllegalStateException("The vehicle engine is not running.");
        }
        isRunning = false;
        System.out.println(brand + " engine stopped.");
    }
    // Method to refuel the vehicle
    public void refuel(int amount)
    {
        if (amount <= 0)
        {
            throw new IllegalArgumentException("Refuel amount must be positive.");
        }
        fuelLevel += amount;
        if (fuelLevel > 100)
        {
            fuelLevel = 100; // Fuel level cannot exceed 100%
        }
        System.out.println(brand + " refueled. Current fuel level: " + fuelLevel + "%");
    }
    // Method to drive the vehicle
    public void drive()
    {
        if (!isRunning)
        {
            throw new IllegalStateException("The vehicle is not running. Start the engine first.");
        }
        System.out.println(brand + " is driving.");
    }
}
```

//Main.java

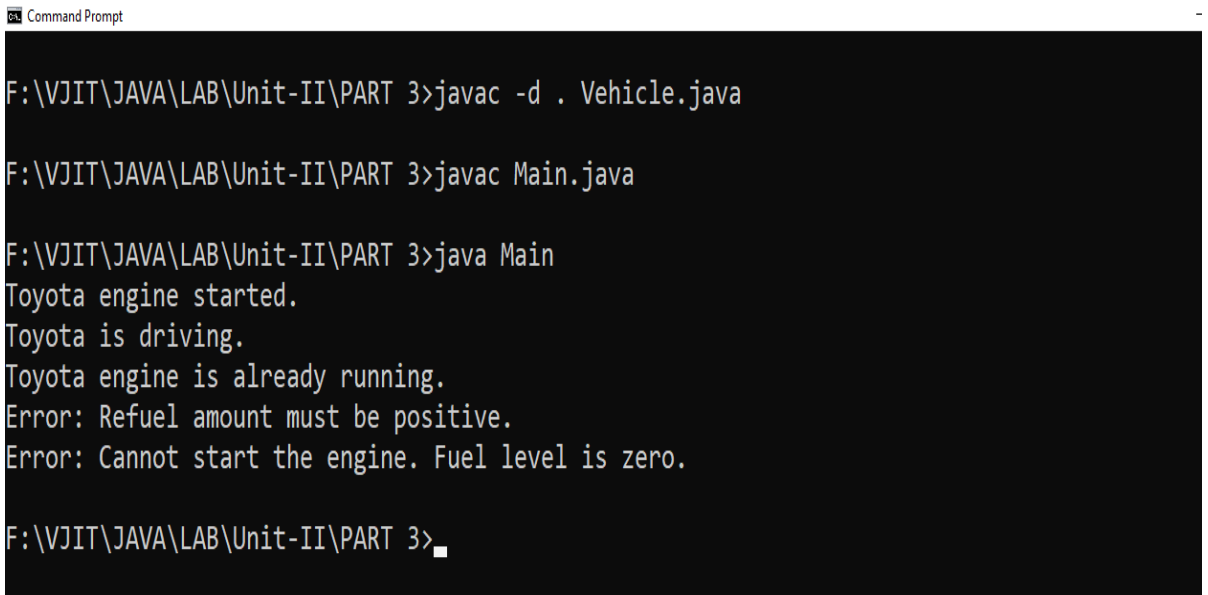
```
import vehicle.Vehicle;
public class Main
{
    public static void main(String[] args)
    {
        try
        {
            // Create a new vehicle with a valid fuel level
            Vehicle v1 = new Vehicle("Toyota", 50);

            // Start the engine
            v1.startEngine();
            v1.drive();
            // Attempt to start the engine again (already running)
            v1.startEngine();
            // Attempt to refuel with a negative value (invalid)
            v1.refuel(-10);
        }
        catch (IllegalArgumentException e)
        {
            System.out.println("Error: " + e.getMessage());
        }
        catch (ArithmeticException e)
        {
            System.out.println("Error: " + e.getMessage());
        }
        catch (IllegalStateException e)
        {
            System.out.println("Error: " + e.getMessage());
        }
        catch (NullPointerException e)
        {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

*// Simulate another scenario where vehicle is stopped*

```
try
{
    // Create another vehicle with zero fuel
    Vehicle v2 = new Vehicle("Ford", 0);
    // Attempt to start the engine without fuel
    v2.startEngine();
}
catch (ArithmeticException e)
{
    System.out.println("Error: " + e.getMessage());
}
}
```

### Output:



```
Command Prompt

F:\VJIT\JAVA\LAB\Unit-II\PART 3>javac -d . Vehicle.java

F:\VJIT\JAVA\LAB\Unit-II\PART 3>javac Main.java

F:\VJIT\JAVA\LAB\Unit-II\PART 3>java Main
Toyota engine started.
Toyota is driving.
Toyota engine is already running.
Error: Refuel amount must be positive.
Error: Cannot start the engine. Fuel level is zero.

F:\VJIT\JAVA\LAB\Unit-II\PART 3>
```

## Usage of try, catch, finally, throw, and throws

In Java, exception handling involves the use of the try, catch, finally, throw, and throws keywords.

<b>try</b> block	This is where you place the code that might throw an exception.
<b>catch</b> block	This block handles exceptions that occur in the try block.
<b>finally</b> block	This block always runs, regardless of whether an exception is thrown or not, and is typically used for cleanup operations.
<b>throw</b> keyword	Used to explicitly throw an exception.
<b>throws</b> keyword	Used in a method signature to declare that the method can throw one or more exceptions.

### 1. try block

- A **try** block consists of all the doubtful statements that can throw exceptions.
- A try block cannot be executed on itself; it requires at least one **catch** block or **finally** block.
- If an **exception** occurs, the control flows from the **try-to-catch** block.
- When an exception occurs in a try block, the appropriate exception object is redirected to the catch block. This **catch** block handles the exception according to its statements and continues the execution.

#### *Syntax*

```
try
{
    //Doubtful Statements.
}
```



## 2. catch block

- The **catch** block handles the exception raised in the **try** block.
- The catch block or blocks follow every try block.
- The catch block catches the **thrown exception** as its parameter and executes the statements inside it.
- The declared exception must be the **parent class exception**, the generated exception type in the **exception class hierarchy**, or a **user-defined exception**.

### *Syntax*

```
try
{
    //Doubtful Statements.
}
catch(Exception e)
{
    // code to handle exceptions
}
```

### Multiple catch Blocks

We can use multiple catch statements for different kinds of exceptions that can occur from a single block of code in the try block.

### *Syntax*

```
try
{
    // code to check exceptions
}
catch (exception1)
{
    // code to handle the exception
}
catch (exception2)
{
    // code to handle the exception
}
```

### 3. finally block

The **finally block in Java** always executes even if there are no **exceptions**. This is an optional block. It is used to execute important statements such as closing statements, releasing resources, and releasing memory. There could be one final block for every try block. This finally block executes after the **try...catch block**.

#### *Syntax*

```
try
{
    //code
}
catch (ExceptionType1 e1)
{
    // catch block
}
finally
{
    // finally block always executes
}
```

### 4. throw Keyword

- The throw keyword is used to explicitly throw a checked or an **unchecked exception**.
- The exception that is thrown needs to be of type Throwable or a subclass of Throwable.
- We can also define our own set of conditions for which we can throw an exception explicitly using the **throw keyword**.
- The program's execution flow stops immediately after the throw statement is executed, and the nearest try block is checked to see if it has a catch statement that matches the type of exception.

#### *Syntax*

```
throw new exception_class("error message");
```

## 5. throws keyword

The **throws keyword** is used in the method signature to indicate that a **method in Java** can throw particular exceptions. This notifies the method that it must manage or propagate these exceptions to the caller.

### *Syntax*

```
return_type method_name() throws exception_class_name
{
    //method code
}
```

Here's an example of how each of these can be used in the context of a Vehicle class.

### **Example: try, catch, finally, throw, and throws in Java**

*// Custom unchecked exception for when the vehicle is already stopped*

class VehicleAlreadyStoppedException extends **RuntimeException**

```
{
    public VehicleAlreadyStoppedException(String message)
    {
        super(message);
    }
}
```

*// Custom checked exception for out of fuel error*

class OutOfFuelException extends **Exception**

```
{
    public OutOfFuelException(String message)
    {
        super(message);
    }
}
```

```
class Vehicle
{
    private boolean isRunning = false;
    private int fuelLevel = 10; // Fuel level out of 100
    // Method to start the vehicle
    public void start()
    {
        System.out.println("Vehicle is starting...");
        isRunning = true;
    }
    // Method to stop the vehicle
    public void stop()
    {
        if (!isRunning)
        {
            // Throw a custom unchecked exception if trying to stop an already stopped vehicle
            throw new VehicleAlreadyStoppedException("Vehicle is already stopped.");
        }
        System.out.println("Vehicle is stopping...");
        isRunning = false;
    }
    // Method to accelerate the vehicle
    public void accelerate(int speed)
    {
        try
        {
            if (speed < 0)
            {
                throw new IllegalArgumentException("Speed cannot be negative.");
            }
            System.out.println("Accelerating to " + speed + " km/h...");
        }
        catch (IllegalArgumentException e)
        {
            System.out.println("Error while accelerating: " + e.getMessage());
        }
        finally
        {
            System.out.println("Acceleration attempt completed.");
        }
    }
}
```

*// Method to drive, might throw an OutOfFuelException*

```
public void drive() throws OutOfFuelException
{
    if (fuelLevel <= 0)
    {
        // Throw a custom checked exception if the vehicle is out of fuel
        throw new OutOfFuelException("Out of fuel! Cannot drive.");
    }
    System.out.println("Vehicle is driving...");
    fuelLevel -= 10; // Decrease fuel level by 10
}
```

public class **VehicleTest**

```
{
    public static void main(String[] args)
    {
        Vehicle v = new Vehicle();
```

*// Example 1: Using try, catch, finally*

```
    try
    {
        v.accelerate(-10); // Invalid speed, will throw IllegalArgumentException
    }
    catch (IllegalArgumentException e)
    {
        System.out.println("Caught exception in accelerate method: " + e.getMessage());
    }
    finally
    {
        System.out.println("Finally block executed for acceleration.");
    }
```

*// Example 2: Using throw to manually throw a custom unchecked exception*

```
    try
    {
        v.stop(); // This will stop the vehicle
        v.stop(); // This will throw VehicleAlreadyStoppedException
    }
```

```
catch (VehicleAlreadyStoppedException e)
{
    System.out.println("Caught exception in stop method: " + e.getMessage());
}
```

*// Example 3: Using throws to declare a custom checked exception*

```
try
{
    v.drive();
    v.drive(); // Might throw OutOfFuelException
}
catch (OutOfFuelException e)
{
    System.out.println("Caught exception in drive method: " + e.getMessage());
}
}
```

### Output:

Command Prompt

```
F:\VJIT\JAVA\LAB\Unit-II\PART 3\UserDefinedExceptions>java VehicleTest
Error while accelerating: Speed cannot be negative.
Acceleration attempt completed.
Finally block executed for acceleration.
Caught exception in stop method: Vehicle is already stopped.
Vehicle is driving...
Caught exception in drive method: Out of fuel! Cannot drive.

F:\VJIT\JAVA\LAB\Unit-II\PART 3\UserDefinedExceptions>
```

## Creating own Exception sub Classes

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes.

- All exceptions must be a child of Throwable.
- If you want to create a checked exception that is automatically enforced you to extend the Exception class.
- If you want to create a unchecked exception, you need to extend the RuntimeException class.

We can define our own **checked Exceptions** class as below –

### *Syntax*

```
class MyException extends Exception
{
    Statements
}
```

You just need to extend the predefined Exception class to create your own checked Exceptions.

We can define our own **unchecked Exceptions** class as below –

### *Syntax*

```
class MyException extends RuntimeException
{
    Statements
}
```

You just need to extend the predefined RuntimeException class to create your own Exception.

*// Custom unchecked exception for when the vehicle is already stopped*

```
class VehicleAlreadyStoppedException extends RuntimeException
{
    public VehicleAlreadyStoppedException(String message)
    {
        super(message);
    }
}
```

*// Custom checked exception for out of fuel error*

```
class OutOfFuelException extends Exception
{
    public OutOfFuelException(String message)
    {
        super(message);
    }
}
```

```
class Vehicle
```

```
{
    private boolean isRunning = true;
    private int fuelLevel = 10; // Fuel level out of 100
```

*// Method to stop the vehicle*

```
public void stop()
{
    if (!isRunning)
    {
        // Throw a custom unchecked exception if trying to stop an already stopped vehicle
        throw new VehicleAlreadyStoppedException("Vehicle is already stopped.");
    }
    System.out.println("Vehicle is stopping...");
    isRunning = false;
}
```



```
// Method to drive, might throw an OutOfFuelException
public void drive() throws OutOfFuelException
{
    if (fuelLevel <= 0)
    {
        // Throw a custom checked exception if the vehicle is out of fuel
        throw new OutOfFuelException("Out of fuel! Cannot drive.");
    }
    System.out.println("Vehicle is driving...");
    fuelLevel -= 10; // Decrease fuel level by 10
}

public class VehicleTestCustomEx
{
    public static void main(String[] args)
    {
        Vehicle v = new Vehicle();

//Using throw to manually throw a custom unchecked exception
        try
        {
            v.stop(); // This will stop the vehicle
            v.stop(); // This will throw VehicleAlreadyStoppedException
        }
        catch (VehicleAlreadyStoppedException e)
        {
            System.out.println("Caught exception in stop method: " + e.getMessage());
        }

//Using throws to declare a custom checked exception
        try
        {
            v.drive();
            v.drive(); // Might throw OutOfFuelException
        }
```

```
        catch (OutOfFuelException e)
        {
            System.out.println("Caught exception in drive method: " + e.getMessage());
        }
    }
}
```

Command Prompt

```
F:\VJIT\JAVA\LAB\Unit-II\PART 3\UserDefinedExceptions>javac VehicleTestCustomEx.java

F:\VJIT\JAVA\LAB\Unit-II\PART 3\UserDefinedExceptions>java VehicleTestCustomEx
Vehicle is stopping...
Caught exception in stop method: Vehicle is already stopped.
Vehicle is driving...
Caught exception in drive method: Out of fuel! Cannot drive.

F:\VJIT\JAVA\LAB\Unit-II\PART 3\UserDefinedExceptions>_
```