
Classes and Objects

Classes

A **class** in Java is a blueprint for creating objects. It defines the properties (fields) and behaviors (methods) that objects of the class will have.

Syntax for defining a class:

```
class ClassName {
    // attributes (fields)
    // methods (behaviors)
}
```

Objects

An **object** is an instance of a class, created using the `new` keyword.

You create an object from a class by invoking its constructor.

```
class Car {
    String make;
    String model;

    // Constructor
    public Car(String make, String model) {
        this.make = make;
        this.model = model;
    }

    // Method
    public void startEngine() {
        System.out.println("Engine started");
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", "Corolla");  // Object creation
        myCar.startEngine();  // Method call
    }
}
```

Constructors

A **constructor** is a special method used to initialize objects of a class. It is called automatically when you create an object.

Constructor Overloading: You can have multiple constructors with different parameters.

Example:

```
class Car {
String make;
String model;

// Constructor with two parameters
public Car(String make, String model) {
      this.make = make;
      this.model = model;
}

// Constructor with no parameters
public Car() {
      this.make = "Unknown";
      this.model = "Unknown";
}

public void display() {
      System.out.println("Car Make: " + make + ", Model: " + model);
}
}

public class Main {
public static void main(String[] args) {
      Car car1 = new Car("Toyota", "Corolla");
      car1.display();  // Output: Car Make: Toyota, Model: Corolla

      Car car2 = new Car();  // Using the default constructor
      car2.display();  // Output: Car Make: Unknown, Model: Unknown
}
}
```

Inheritance

Inheritance allows a class (subclass) to inherit fields and methods from another class (superclass).

In Java, the `extends` keyword is used to indicate inheritance.

Types of Inheritance in Java

Single Inheritance: A subclass inherits from a single superclass.

Multilevel Inheritance: A class inherits from another class, which in turn inherits from another class.

Hierarchical Inheritance: Multiple classes inherit from a single superclass.

Example (Single Inheritance):

```
class Animal {
public void sound() {
      System.out.println("Animal makes a sound");
}
}

class Dog extends Animal {  // Dog inherits from Animal
public void sound() {
```

```
        System.out.println("Dog barks");
}
}

public class Main {
public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound();  // Output: Dog barks
}
}
```

Example (Multilevel Inheritance):

```
class Animal {
public void sound() {
        System.out.println("Animal makes a sound");
}
}

class Dog extends Animal {
public void sound() {
        System.out.println("Dog barks");
}
}

class Bulldog extends Dog {
public void sound() {
        System.out.println("Bulldog growls");
}
}

public class Main {
public static void main(String[] args) {
        Bulldog bulldog = new Bulldog();
        bulldog.sound();  // Output: Bulldog growls
}
}
```

Polymorphism

Polymorphism allows a method to perform different actions based on the object that invokes it.

Types of Polymorphism in Java

Compile-time Polymorphism (Method Overloading): The method name is the same, but the parameters differ.

Runtime Polymorphism (Method Overriding): A subclass provides a specific implementation of a method that is already defined in its superclass.

Example (Method Overloading - Compile-time Polymorphism):

```
class Calculator {
public int add(int a, int b) {
        return a + b;
}
```

```
public double add(double a, double b) {
      return a + b;
}
}

public class Main {
public static void main(String[] args) {
      Calculator calc = new Calculator();
      System.out.println(calc.add(2, 3));       // Output: 5 (int
      version)
      System.out.println(calc.add(2.5, 3.5));   // Output: 6.0 (double
      version)
}
}
```

Example (Method Overriding - Runtime Polymorphism):

```
class Animal {
public void sound() {
      System.out.println("Animal makes a sound");
}
}

class Dog extends Animal {
@Override
public void sound() {
      System.out.println("Dog barks");
}
}

public class Main {
public static void main(String[] args) {
      Animal myDog = new Dog();  // Reference to Animal, but object of
      Dog
      myDog.sound();  // Output: Dog barks (runtime polymorphism)
}
}
```

Abstraction

Abstraction is the process of hiding the implementation details and showing only the functionality to the user. It is achieved using **abstract classes** and **interfaces**.

Abstract Class:

An **abstract class** cannot be instantiated directly. It can have both abstract (without body) and non-abstract (with body) methods.

A subclass must implement all abstract methods of the abstract class.

Example (Abstract Class):

```
abstract class Animal {
public abstract void sound();  // Abstract method

public void sleep() {  // Non-abstract method
```

```
      System.out.println("Animal is sleeping");
}
}

class Dog extends Animal {
@Override
public void sound() {
      System.out.println("Dog barks");
}
}

public class Main {
public static void main(String[] args) {
      Dog dog = new Dog();
      dog.sound();  // Output: Dog barks
      dog.sleep();  // Output: Animal is sleeping
}
}
```

Interface:

An **interface** is a contract that a class must adhere to. It contains only abstract methods (until Java 8, after which default and static methods can also be included).

Example (Interface):

```
interface Animal {
void sound();  // Abstract method
}

class Dog implements Animal {  // Dog implements Animal interface
@Override
public void sound() {
      System.out.println("Dog barks");
}
}

public class Main {
public static void main(String[] args) {
      Animal dog = new Dog();
      dog.sound();  // Output: Dog barks
}
}
```

Encapsulation

Encapsulation is the concept of bundling the data (fields) and methods that operate on the data into a single unit (class). It also involves restricting access to certain details.

Access Modifiers:

public: Accessible from any other class.

private: Accessible only within the class.

protected: Accessible within the class, subclasses, and classes in the same package.

default: Accessible within classes in the same package (no modifier).

Example (Encapsulation with Access Modifiers):

```
class Car {
private String make;
private String model;

// Getter and Setter methods
public String getMake() {
      return make;
}

public void setMake(String make) {
      this.make = make;
}

public String getModel() {
      return model;
}

public void setModel(String model) {
      this.model = model;
}
}

public class Main {
public static void main(String[] args) {
      Car car = new Car();
      car.setMake("Toyota");  // Using setter to set private attribute
      car.setModel("Corolla");

      System.out.println("Car Make: " + car.getMake());  // Using getter
      to access private attribute
      System.out.println("Car Model: " + car.getModel());
}
}
```
