Java Classes and Objects - Complete Beginner's Guide

Table of Contents

Introduction to Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to design applications. Java is an object-oriented language, which means everything in Java is associated with classes and objects.

Key Concepts:

- **Class**: A blueprint or template for creating objects
- **Object**: An instance of a class
- **Encapsulation**: Bundling data and methods together
- **Inheritance**: Creating new classes based on existing classes
- **Polymorphism**: Using one interface for different data types

What is a Class?

A **class** is a blueprint or template that defines the structure and behavior of objects. It doesn't consume memory until an object is created from it.

Class Structure:

```
java
```

```
public class ClassName {
    // Instance variables (attributes)
    private dataType variableName;

    // Constructor
    public ClassName() {
        // initialization code
    }

    // Methods (behaviors)
    public returnType methodName() {
        // method body
    }
}
```

Example:

java

```
public class Car {
    // Instance variables
    private String brand;
    private String model;
    private int year;
    private double price;
}
```

What is an Object?

An **object** is an instance of a class. When you create an object, you're creating a specific instance with actual values for the class attributes.

Key Points:

- Objects have **state** (values of instance variables)
- Objects have **behavior** (methods they can perform)
- Each object has its own copy of instance variables
- Objects are created in heap memory

Creating Objects:

java

```
ClassName objectName = new ClassName();
```

Example:

```java
Car myCar = new Car();
Car yourCar = new Car();
```

Creating Classes and Objects

Let's create a complete example:

```java
public class Student {
    // Instance variables
    private String name;
    private int age;
    private String studentId;
    private double gpa;

    // We'll add methods here later
}

// Creating objects
public class Main {
    public static void main(String[] args) {
        Student student1 = new Student();
        Student student2 = new Student();
    }
}
```

Instance Variables and Methods

Instance Variables:

- Belong to each object individually
- Each object has its own copy
- Declared inside the class but outside methods
- Usually declared as private (encapsulation)

Instance Methods:

- Operate on instance variables
- Can access and modify object's state
- Called using object reference

```java
public class Rectangle {
    private double length;
    private double width;

    // Instance method
    public double calculateArea() {
        return length * width;
    }

    // Instance method
    public double calculatePerimeter() {
        return 2 * (length + width);
    }
}
```

The 'this' Keyword

The `this` keyword is a reference to the current object. It's used to refer to the current object's instance variables and methods.

Uses of 'this':

1. Distinguishing between instance variables and parameters:

```java
public class Person {
    private String name;
    private int age;

    public void setName(String name) {
        this.name = name; // this.name refers to instance variable
                          // name refers to parameter
    }

    public void setAge(int age) {
        this.age = age; // Without 'this', it would be ambiguous
    }
}
```

2. Calling other constructors (constructor chaining):

```java
```

```java
public class Employee {
    private String name;
    private int id;
    private double salary;

    // Default constructor
    public Employee() {
        this("Unknown", 0, 0.0); // Calls parameterized constructor
    }

    // Parameterized constructor
    public Employee(String name, int id, double salary) {
        this.name = name;
        this.id = id;
        this.salary = salary;
    }
}
```

3. Calling other methods:

```java
public class Calculator {
    public void performCalculation() {
        this.displayResult(); // Explicit call using 'this'
        displayResult();      // Implicit call (same as above)
    }

    private void displayResult() {
        System.out.println("Calculation completed");
    }
}
```

4. Returning current object:

```java
public class Builder {
    private String data;

    public Builder setData(String data) {
        this.data = data;
        return this; // Returns current object for method chaining
    }
}
```

Constructors

A **constructor** is a special method that's automatically called when an object is created. It's used to initialize the object's state.

Constructor Rules:

1. Name must be same as class name
2. No return type (not even void)
3. Called automatically when object is created
4. Can be overloaded (multiple constructors)

Types of Constructors:

1. Default Constructor:

```java
public class Book {
    private String title;
    private String author;

    // Default constructor
    public Book() {
        title = "Unknown";
        author = "Unknown";
        System.out.println("Default constructor called");
    }
}
```

2. Parameterized Constructor:

```java
public class Book {
    private String title;
    private String author;

    // Parameterized constructor
    public Book(String title, String author) {
        this.title = title;
        this.author = author;
        System.out.println("Parameterized constructor called");
    }
}
```

3. Constructor Overloading:

java

```java
public class Student {
    private String name;
    private int age;
    private String course;

    // Default constructor
    public Student() {
        this.name = "Unknown";
        this.age = 0;
        this.course = "Not Enrolled";
    }

    // Constructor with name
    public Student(String name) {
        this.name = name;
        this.age = 0;
        this.course = "Not Enrolled";
    }

    // Constructor with name and age
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
        this.course = "Not Enrolled";
    }

    // Constructor with all parameters
    public Student(String name, int age, String course) {
        this.name = name;
        this.age = age;
        this.course = course;
    }
}
```

4. Copy Constructor (Manual Implementation):

java

```java
public class Point {
    private int x;
    private int y;

    // Regular constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // Copy constructor
    public Point(Point other) {
        this.x = other.x;
        this.y = other.y;
    }
}
```

Destructors (Finalize Method)

Java doesn't have destructors like C++, but it has a **finalize()** method that can be overridden. However, it's rarely used and not recommended in modern Java.

The finalize() Method:

java

```java
public class ResourceManager {
    private String resourceName;

    public ResourceManager(String name) {
        this.resourceName = name;
        System.out.println("Resource " + name + " acquired");
    }

    // finalize method (called by garbage collector)
    @Override
    protected void finalize() throws Throwable {
        try {
            System.out.println("Resource " + resourceName + " is being cleaned up");
            // Cleanup code here
        } finally {
            super.finalize();
        }
    }
}
```

Modern Alternative - try-with-resources:

java

```java
public class ModernResource implements AutoCloseable {
    private String name;

    public ModernResource(String name) {
        this.name = name;
        System.out.println("Resource " + name + " acquired");
    }

    @Override
    public void close() {
        System.out.println("Resource " + name + " closed");
    }

    // Usage with try-with-resources
    public static void main(String[] args) {
        try (ModernResource resource = new ModernResource("Database")) {
            // Use resource
        } // Automatically calls close()
    }
}
```

Getters and Setters

Getters and **Setters** are methods used to access and modify private instance variables. This implements the principle of **encapsulation**.

Why Use Getters and Setters?

1. **Data Hiding**: Keep instance variables private
2. **Validation**: Add validation logic when setting values
3. **Control**: Control how data is accessed and modified
4. **Flexibility**: Can change internal implementation without affecting client code

Basic Getter and Setter:

java

```java
public class Person {
    private String name;
    private int age;

    // Getter for name
    public String getName() {
        return name;
    }

    // Setter for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter for age
    public int getAge() {
        return age;
    }

    // Setter for age
    public void setAge(int age) {
        this.age = age;
    }
}
```

Getters and Setters with Validation:

```java
```

```
public class BankAccount {
    private String accountNumber;
    private double balance;
    private String accountHolder;

    // Getter for account number (read-only)
    public String getAccountNumber() {
        return accountNumber;
    }

    // Getter for balance
    public double getBalance() {
        return balance;
    }

    // Setter for balance with validation
    public void setBalance(double balance) {
        if (balance < 0) {
            throw new IllegalArgumentException("Balance cannot be negative");
        }
        this.balance = balance;
    }

    // Getter for account holder
    public String getAccountHolder() {
        return accountHolder;
    }

    // Setter for account holder with validation
    public void setAccountHolder(String accountHolder) {
        if (accountHolder == null || accountHolder.trim().isEmpty()) {
            throw new IllegalArgumentException("Account holder name cannot be null or empty");
        }
        this.accountHolder = accountHolder.trim();
    }

    // Method to deposit money
    public void deposit(double amount) {
        if (amount <= 0) {
            throw new IllegalArgumentException("Deposit amount must be positive");
        }
        this.balance += amount;
    }

    // Method to withdraw money
    public boolean withdraw(double amount) {
```

```
        if (amount <= 0) {
            throw new IllegalArgumentException("Withdrawal amount must be positive");
        }
        if (amount > balance) {
            return false; // Insufficient funds
        }
        this.balance -= amount;
        return true;
    }
}
```

Advanced Setter with Business Logic:

java

```
public class Employee {
    private String name;
    private double salary;
    private String department;
    private Date hireDate;

    // Setter with business logic
    public void setSalary(double salary) {
        if (salary < 0) {
            throw new IllegalArgumentException("Salary cannot be negative");
        }
        if (salary > 1000000) {
            System.out.println("High salary detected. Requires approval.");
        }
        this.salary = salary;
    }

    // Computed property (getter only)
    public double getAnnualSalary() {
        return salary * 12;
    }

    // Getter with formatting
    public String getFormattedSalary() {
        return String.format("$%.2f", salary);
    }
}
```

Access Modifiers

Access modifiers control the visibility of classes, methods, and variables.

Types of Access Modifiers:

1. **private**: Accessible only within the same class

```java
public class Example {
    private int privateVar = 10;

    private void privateMethod() {
        System.out.println("Private method");
    }
}
```

2. **default (package-private)**: Accessible within the same package

```java
class Example {
    int packageVar = 20;

    void packageMethod() {
        System.out.println("Package method");
    }
}
```

3. **protected**: Accessible within same package and subclasses

```java
public class Example {
    protected int protectedVar = 30;

    protected void protectedMethod() {
        System.out.println("Protected method");
    }
}
```

4. **public**: Accessible from anywhere

```java
```

```java
public class Example {
    public int publicVar = 40;

    public void publicMethod() {
        System.out.println("Public method");
    }
}
```

Complete Example

Here's a comprehensive example that demonstrates all concepts:

java

```java
// Complete Car class demonstrating all concepts
public class Car {
    // Private instance variables (encapsulation)
    private String brand;
    private String model;
    private int year;
    private double price;
    private String color;
    private boolean isRunning;
    private double mileage;

    // Static variable (belongs to class, not objects)
    private static int totalCarsCreated = 0;

    // Default constructor
    public Car() {
        this("Unknown", "Unknown", 2020, 0.0, "White");
        System.out.println("Default constructor called");
    }

    // Parameterized constructor with validation
    public Car(String brand, String model, int year, double price, String color) {
        setBrand(brand);
        setModel(model);
        setYear(year);
        setPrice(price);
        setColor(color);
        this.isRunning = false;
        this.mileage = 0.0;
        totalCarsCreated++;
        System.out.println("Parameterized constructor called");
    }

    // Copy constructor
    public Car(Car other) {
        this(other.brand, other.model, other.year, other.price, other.color);
        this.mileage = other.mileage;
        System.out.println("Copy constructor called");
    }

    // Getters
    public String getBrand() {
        return brand;
    }

    public String getModel() {
```

```
    return model;
}

public int getYear() {
    return year;
}

public double getPrice() {
    return price;
}

public String getColor() {
    return color;
}

public boolean isRunning() {
    return isRunning;
}

public double getMileage() {
    return mileage;
}

// Setters with validation
public void setBrand(String brand) {
    if (brand == null || brand.trim().isEmpty()) {
        throw new IllegalArgumentException("Brand cannot be null or empty");
    }
    this.brand = brand.trim();
}

public void setModel(String model) {
    if (model == null || model.trim().isEmpty()) {
        throw new IllegalArgumentException("Model cannot be null or empty");
    }
    this.model = model.trim();
}

public void setYear(int year) {
    int currentYear = java.time.Year.now().getValue();
    if (year < 1900 || year > currentYear + 1) {
        throw new IllegalArgumentException("Invalid year: " + year);
    }
    this.year = year;
}

public void setPrice(double price) {
```

```
    if (price < 0) {
        throw new IllegalArgumentException("Price cannot be negative");
    }
    this.price = price;
}

public void setColor(String color) {
    if (color == null || color.trim().isEmpty()) {
        this.color = "White"; // Default color
    } else {
        this.color = color.trim();
    }
}

// Business methods
public void startEngine() {
    if (isRunning) {
        System.out.println(this.brand + " " + this.model + " is already running!");
    } else {
        this.isRunning = true;
        System.out.println(this.brand + " " + this.model + " engine started!");
    }
}

public void stopEngine() {
    if (!isRunning) {
        System.out.println(this.brand + " " + this.model + " is already stopped!");
    } else {
        this.isRunning = false;
        System.out.println(this.brand + " " + this.model + " engine stopped!");
    }
}

public void drive(double miles) {
    if (!isRunning) {
        System.out.println("Cannot drive. Engine is not running!");
        return;
    }
    if (miles <= 0) {
        System.out.println("Invalid distance!");
        return;
    }
    this.mileage += miles;
    System.out.println("Drove " + miles + " miles. Total mileage: " + this.mileage);
}

// Computed properties
```

```
    public int getAge() {
        return java.time.Year.now().getValue() - this.year;
    }

    public String getFullName() {
        return this.brand + " " + this.model;
    }

    public double getDepreciatedValue() {
        int age = getAge();
        double depreciationRate = 0.15; // 15% per year
        return this.price * Math.pow(1 - depreciationRate, age);
    }

    // Static methods
    public static int getTotalCarsCreated() {
        return totalCarsCreated;
    }

    // toString method for string representation
    @Override
    public String toString() {
        return String.format("Car{brand='%s', model='%s', year=%d, price=%.2f, color='%s', mileage=%.1f, running=%s}",
                brand, model, year, price, color, mileage, isRunning);
    }

    // equals method for object comparison
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;

        Car car = (Car) obj;
        return year == car.year &&
                Double.compare(car.price, price) == 0 &&
                brand.equals(car.brand) &&
                model.equals(car.model) &&
                color.equals(car.color);
    }

    // finalize method (not recommended in modern Java)
    @Override
    protected void finalize() throws Throwable {
        try {
            System.out.println("Car object is being garbage collected: " + getFullName());
        } finally {
            super.finalize();
```

```
        }
    }
}

// Demo class to show usage
public class CarDemo {
    public static void main(String[] args) {
        System.out.println("=== Car Class Demo ===");

        // Creating objects using different constructors
        Car car1 = new Car(); // Default constructor
        Car car2 = new Car("Toyota", "Camry", 2022, 25000.0, "Blue"); // Parameterized
        Car car3 = new Car(car2); // Copy constructor

        System.out.println("\nTotal cars created: " + Car.getTotalCarsCreated());

        // Using setters
        car1.setBrand("Honda");
        car1.setModel("Civic");
        car1.setYear(2023);
        car1.setPrice(22000.0);
        car1.setColor("Red");

        // Using getters
        System.out.println("\nCar 1 Details:");
        System.out.println("Brand: " + car1.getBrand());
        System.out.println("Model: " + car1.getModel());
        System.out.println("Full Name: " + car1.getFullName());
        System.out.println("Age: " + car1.getAge() + " years");
        System.out.println("Depreciated Value: $" + String.format("%.2f", car1.getDepreciatedValue()));

        // Using business methods
        System.out.println("\n=== Car Operations ===");
        car1.startEngine();
        car1.drive(100.5);
        car1.drive(50.0);
        car1.stopEngine();

        // Display all car objects
        System.out.println("\n=== All Cars ===");
        System.out.println("Car 1: " + car1);
        System.out.println("Car 2: " + car2);
        System.out.println("Car 3: " + car3);

        // Testing validation
        try {
            Car invalidCar = new Car("", "Model", 2025, -1000, "");
```

```
        } catch (IllegalArgumentException e) {
            System.out.println("\nValidation error: " + e.getMessage());
        }
    }
}
```

Best Practices

1. **Encapsulation:**

- Keep instance variables private
- Provide public getters and setters when needed
- Add validation in setters

2. **Constructor Guidelines:**

- Always provide a default constructor if possible
- Use constructor chaining with `this()`
- Initialize all instance variables
- Add validation in constructors

3. **Method Design:**

- Keep methods focused on single responsibility
- Use meaningful method names
- Add proper validation and error handling

4. **Use of 'this' keyword:**

- Use `this` to resolve naming conflicts
- Use `this()` for constructor chaining
- Be explicit when it improves readability

5. **General Guidelines:**

- Follow naming conventions (camelCase for variables and methods)
- Add proper documentation and comments
- Override `toString()`, `equals()`, and `hashCode()` when needed
- Avoid finalize() method in modern Java
- Use try-with-resources for resource management

6. **Common Patterns:**

java

```java
// Builder Pattern for complex objects
public class CarBuilder {
    private String brand;
    private String model;
    private int year;

    public CarBuilder setBrand(String brand) {
        this.brand = brand;
        return this;
    }

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

    public CarBuilder setYear(int year) {
        this.year = year;
        return this;
    }

    public Car build() {
        return new Car(brand, model, year, 0.0, "White");
    }
}

// Usage:
Car car = new CarBuilder()
    .setBrand("BMW")
    .setModel("X5")
    .setYear(2023)
    .build();
```

This comprehensive guide covers all the fundamental concepts of Java classes and objects. Practice creating your own classes with different scenarios to master these concepts!