📕 Book 1: Basic Java OOP Concepts

A Conversational Guide for Java Freshers

Chapter 1: What is OOP and Why Java?

Student–Expert Dialogue

Student: I keep hearing that Java is an “Object-Oriented Programming” language. What does that actually mean?

Expert: Great question! Let me break it down. Object-Oriented Programming (OOP) is a programming paradigm—a way of thinking about and organizing code. Instead of writing programs as a list of instructions (procedural programming), OOP organizes code around “objects” that represent real-world things.

Student: Can you give me a real-world example?

Expert: Absolutely! Think about a car. A car has: - **Properties**: color, brand, speed, fuel level - **Behaviors**: start(), stop(), accelerate(), brake()

In OOP, we create a “Car” blueprint (called a **class**), and then create actual car instances (called **objects**) from that blueprint.

```java
// This is a Class - a blueprint
class Car {
    // Properties (also called fields or attributes)
    String color;
    String brand;
    int speed;

    // Behaviors (also called methods)
    void start() {
        System.out.println("Car is starting...");
    }

    void accelerate() {
        speed += 10;
        System.out.println("Speed is now: " + speed);
    }
}
// Creating objects (actual cars)
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();  // Creating an object
        myCar.color = "Red";
        myCar.brand = "Toyota";
        myCar.start();
        myCar.accelerate();
    }
}
```

Line-by-line explanation: - `class Car` - Defines a blueprint for cars - `String color;` - Declares a property that every car will have - `void start()` - Defines a behavior (method) that cars can perform - `Car myCar = new Car();` - Creates an actual car object from the blueprint - `myCar.color = "Red";` - Sets the color property for this specific car

Student: Why is Java called an OOP language specifically?

Expert: Java is designed from the ground up with OOP principles. Almost everything in Java is an object (except primitive types like int, char, etc.). Java enforces OOP through:

1. **Classes and Objects** - The fundamental building blocks
2. **Encapsulation** - Hiding internal details
3. **Inheritance** - Reusing code through parent-child relationships
4. **Polymorphism** - One interface, many implementations
5. **Abstraction** - Showing only essential features

Student: What's the benefit of using OOP?

Expert: Excellent question! Here are the key benefits:

```
Benefits of OOP:
┌─────────────────────────────────────────────┐
│ 1. Modularity                               │
│    → Code is organized in separate classes  │
│                                             │
│ 2. Reusability                              │
│    → Write once, use many times             │
│                                             │
│ 3. Maintainability                          │
│    → Easy to update and fix                 │
│                                             │
│ 4. Scalability                              │
│    → Easy to add new features               │
│                                             │
│ 5. Real-world modeling                      │
│    → Code mirrors real-world entities       │
└─────────────────────────────────────────────┘
```

Interview Trap Alert! 🚨

Interviewer might ask: "Is Java 100% object-oriented?"

Wrong Answer: "Yes, Java is completely object-oriented."

Correct Answer: "No, Java is not 100% object-oriented because it uses primitive data types (int, char, boolean, etc.) which are not objects. However, Java provides wrapper classes (Integer, Character, Boolean) to treat primitives as objects when needed."

```java
// Primitive - NOT an object
int age = 25;

// Wrapper class - IS an object
Integer ageObject = 25;  // Auto-boxing
```

Chapter Recap

✅ OOP organizes code around objects that represent real-world entities
✅ Java is fundamentally an OOP language
✅ Four main OOP principles: Encapsulation, Inheritance, Polymorphism, Abstraction
✅ Java is NOT 100% OOP due to primitive types

Interview Questions

1. **Q:** What is Object-Oriented Programming?
   A: A programming paradigm that organizes code around objects containing data (attributes) and code (methods).

2. **Q:** Why use OOP instead of procedural programming?
   A: OOP provides better code organization, reusability, maintainability, and models real-world scenarios more naturally.

3. **Q:** Name the four pillars of OOP.
   A: Encapsulation, Inheritance, Polymorphism, and Abstraction.

Chapter 2: Class and Object

Student–Expert Dialogue

Student: I'm still a bit confused about the difference between a class and an object. Can you clarify?

Expert: Think of it this way: - **Class** = Blueprint/Template (like an architectural plan) - **Object** = Actual instance (like an actual building built from that plan)

Let me show you with code:

```java
// Class - This is the BLUEPRINT
class Student {
    // Instance variables (each student will have their own values)
    String name;
    int rollNumber;
    double marks;

    // Method - behavior
    void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Roll Number: " + rollNumber);
        System.out.println("Marks: " + marks);
    }
}
// Creating objects - These are ACTUAL STUDENTS
public class Main {
    public static void main(String[] args) {
        // Object 1
        Student student1 = new Student();
        student1.name = "Alice";
        student1.rollNumber = 101;
        student1.marks = 85.5;

        // Object 2
        Student student2 = new Student();
        student2.name = "Bob";
        student2.rollNumber = 102;
        student2.marks = 90.0;

        // Each object has its own data
        student1.displayInfo();
        System.out.println("---");
        student2.displayInfo();
    }
}
```

Output:

```
Name: Alice
Roll Number: 101
Marks: 85.5
---
Name: Bob
Roll Number: 102
Marks: 90.0
```

Student: So one class can create multiple objects?

Expert: Exactly! That's the power of OOP. One blueprint, infinite instances.

```
Class vs Object Visualization:

     Class (Student)
    ┌─────────────────┐
    │ name            │  ← Blueprint
    │ rollNumber      │
    │ marks           │
    │ displayInfo()   │
    └─────────────────┘
           │
           │ Creates instances
           ▼
    ┌──────────┬──────────┬──────────┐
    │ Alice    │ Bob      │ Charlie  │  ← Objects
    │ 101      │ 102      │ 103      │
    │ 85.5     │ 90.0     │ 78.0     │
    └──────────┴──────────┴──────────┘
```

Student: What's the `new` keyword doing?

Expert: Great observation! The `new` keyword does three things:

```
Student student1 = new Student();
//       |          |   |
//       |          |   └─ 3. Calls constructor
//       |          └──── 2. Allocates memory
//       └─────────────── 1. Reference variable
```

1. **Allocates memory** in the heap for the object
2. **Calls the constructor** (we'll cover this soon)
3. **Returns a reference** to the newly created object

Student: What are instance variables?

Expert: Instance variables are variables declared inside a class but outside any method. Each object gets its own copy.

```
class BankAccount {
    // Instance variables - each account has its own values
    String accountNumber;
    String holderName;
    double balance;

    void deposit(double amount) {
        balance += amount;  // Modifies THIS object's balance
    }
}
public class Main {
    public static void main(String[] args) {
        BankAccount account1 = new BankAccount();
        account1.balance = 1000;

        BankAccount account2 = new BankAccount();
        account2.balance = 5000;

        account1.deposit(500);  // Only account1's balance changes

        System.out.println("Account 1: " + account1.balance);  // 1500
        System.out.println("Account 2: " + account2.balance);  // 5000
    }
}
```

Student: Can I have methods inside a class?

Expert: Absolutely! Methods define the behaviors of objects. Let me show you different types:

```java
class Calculator {
    // Instance variable
    String brand = "Casio";

    // Method with no parameters and no return
    void displayBrand() {
        System.out.println("Brand: " + brand);
    }

    // Method with parameters and return type
    int add(int a, int b) {
        return a + b;
    }

    // Method with parameters, no return
    void printSum(int a, int b) {
        System.out.println("Sum: " + (a + b));
    }

    // Method with no parameters but has return
    String getBrand() {
        return brand;
    }
}
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();

        calc.displayBrand();              // No return, no params

        int result = calc.add(5, 3);      // Has return, has params
        System.out.println(result);        // 8

        calc.printSum(10, 20);            // No return, has params

        String brandName = calc.getBrand(); // Has return, no params
        System.out.println(brandName);
    }
}
```

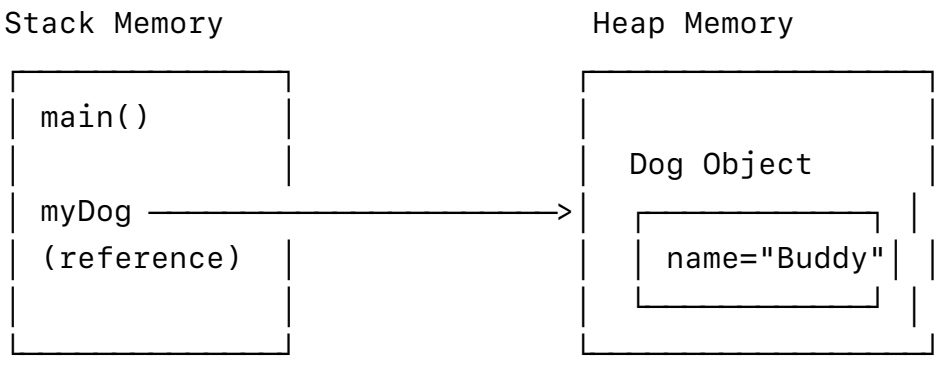
Memory Representation

Student: Where are objects stored in memory?

Expert: Great question! Understanding memory is crucial for interviews.

```java
class Dog {
    String name;
}
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.name = "Buddy";
    }
}
```

Memory Layout:



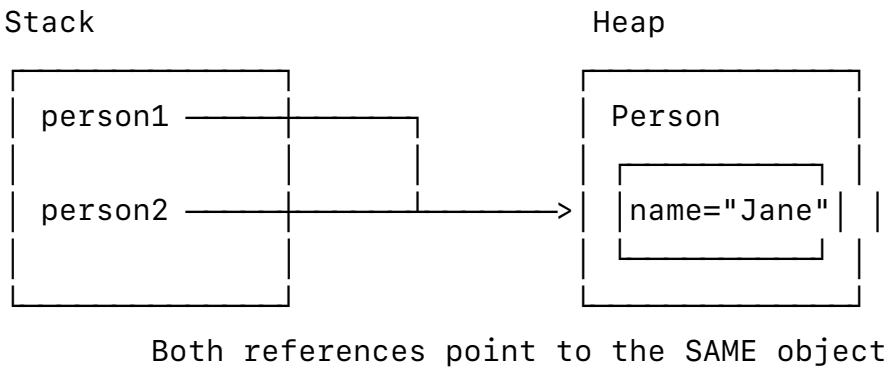

```
Stack: Stores method calls and references
Heap: Stores actual objects
```

Student: What happens if I create multiple references to the same object?

Expert: Excellent question! Let me show you:

```java
class Person {
    String name;
}
public class Main {
    public static void main(String[] args) {
        Person person1 = new Person();
        person1.name = "John";

        Person person2 = person1;  // Both point to same object!

        person2.name = "Jane";

        System.out.println(person1.name);  // Jane
        System.out.println(person2.name);  // Jane
    }
}
```

Memory Visualization:



Interview Trap Alert! 🚨

Interviewer might ask: "What's the difference between these two?"

```java
// Scenario 1
Dog dog1 = new Dog();
Dog dog2 = new Dog();
dog1.name = "Max";
dog2.name = "Max";

// Scenario 2
Dog dog1 = new Dog();
Dog dog2 = dog1;
dog1.name = "Max";
```

Answer: - **Scenario 1**: Two separate objects in memory, both happen to have the same name - **Scenario 2**: One object with two references pointing to it

```java
// Proof:
System.out.println(dog1 == dog2);
// Scenario 1: false (different objects)
// Scenario 2: true (same object)
```

Chapter Recap

✅ Class is a blueprint; Object is an instance
✅ `new` keyword allocates memory and creates objects
✅ Instance variables are unique to each object
✅ Methods define object behaviors
✅ Multiple references can point to the same object

Interview Questions

1. **Q:** What is a class?
   A: A class is a blueprint or template that defines the structure and behavior of objects.

2. **Q:** What is an object?
   A: An object is an instance of a class that exists in memory with its own state (data) and behavior (methods).

3. **Q:** How many objects can be created from one class?
   A: Unlimited. A class is a template that can create any number of objects.

4. **Q:** What does the `new` keyword do?

A: It allocates memory for an object, initializes it, and returns a reference to it.

5. **Q:** What are instance variables?
   A: Variables declared in a class but outside methods. Each object has its own copy.

Chapter 3: Encapsulation

Student–Expert Dialogue

Student: I've heard the term "encapsulation" a lot. What exactly is it?

Expert: Encapsulation is one of the four pillars of OOP. Think of it as "data hiding" or "bundling data with methods." The idea is to: 1. Keep data (variables) private 2. Provide public methods to access/modify that data 3. Control how data is accessed and modified

Student: Why would I want to hide data?

Expert: Let me show you a problem first, then the solution:

```java
// WITHOUT Encapsulation - BAD PRACTICE
class BankAccount {
    public String accountNumber;
    public double balance;  // Anyone can modify this!
}
public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        account.balance = 1000;

        // Problem: Anyone can do this!
        account.balance = -5000;  // Negative balance? That's wrong!

        System.out.println("Balance: " + account.balance);  // -5000
    }
}
```

Student: I see the problem! We can't control what values are set.

Expert: Exactly! Now let's fix this with encapsulation:

```java
// WITH Encapsulation - GOOD PRACTICE
class BankAccount {
    // Private variables - cannot be accessed directly from outside
    private String accountNumber;
    private double balance;

    // Public getter - allows reading the value
    public double getBalance() {
        return balance;
    }

    // Public setter - allows setting the value WITH VALIDATION
    public void setBalance(double balance) {
        if (balance >= 0) {
            this.balance = balance;
        } else {
            System.out.println("Error: Balance cannot be negative!");
        }
    }

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

    // Setter for account number
    public void setAccountNumber(String accountNumber) {
        this.accountNumber = accountNumber;
    }

    // Business logic methods
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposited: " + amount);
        }
    }

    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            System.out.println("Withdrawn: " + amount);
        } else {
            System.out.println("Insufficient balance or invalid amount!");
```

```
        }
    }
}
public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();

        // Cannot do this anymore:
        // account.balance = -5000;  // Compile error!

        // Must use setters
        account.setBalance(1000);
        account.setBalance(-500);  // Validation prevents this

        // Use getters to read
        System.out.println("Balance: " + account.getBalance());
// 1000

        // Use business methods
        account.deposit(500);
        account.withdraw(200);
        System.out.println("Final Balance: " +
account.getBalance());  // 1300
    }
}
```

Output:

```
Error: Balance cannot be negative!
Balance: 1000.0
Deposited: 500.0
Withdrawn: 200.0
Final Balance: 1300.0
```

Student: So getters and setters are the key to encapsulation?

Expert: They're the mechanism, but the principle is broader. Let me break it down:

```
Encapsulation Principles:
┌──────────────────────────────────────────┐
│ 1. Make fields private                   │
│    private String name;                  │
│                                          │
│ 2. Provide public getters                │
│    public String getName()               │
│                                          │
│ 3. Provide public setters (with logic)   │
│    public void setName(String name)      │
│                                          │
│ 4. Add validation in setters             │
│    if (name != null && !name.isEmpty())  │
└──────────────────────────────────────────┘
```

Student: Do I always need both getter and setter?

Expert: No! That's a common misconception. You provide only what's needed:

```java
class Employee {
    private String name;
    private final String employeeId;  // Cannot change after creation
    private double salary;

    // Constructor
    public Employee(String name, String employeeId, double salary) {
        this.name = name;
        this.employeeId = employeeId;
        this.salary = salary;
    }

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

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

```

```java
    // Only getter for employeeId (read-only)
    public String getEmployeeId() {
        return employeeId;
    }
    // No setter - employeeId cannot be changed!

    // No getter for salary (private information)
    // But we can have a method to increase it
    public void increaseSalary(double percentage) {
        if (percentage > 0 && percentage <= 20) {
            salary += salary * (percentage / 100);
        }
    }

    // Method to check if salary is above threshold (without revealing exact salary)
    public boolean isHighEarner() {
        return salary > 100000;
    }
}
public class Main {
    public static void main(String[] args) {
        Employee emp = new Employee("John", "E001", 80000);

        // Can change name
        emp.setName("John Doe");

        // Can read employee ID
        System.out.println("ID: " + emp.getEmployeeId());

        // Cannot change employee ID
        // emp.setEmployeeId("E002");  // No such method!

        // Cannot read exact salary
        // System.out.println(emp.getSalary());  // No such method!

        // But can check if high earner
        System.out.println("High earner? " + emp.isHighEarner()); // false

        // Can increase salary
        emp.increaseSalary(10);
        System.out.println("High earner? " + emp.isHighEarner()); // false
    }
}
```

Student: What about validation? Can you show more examples?

Expert: Absolutely! Validation is where encapsulation really shines:

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

    // Name validation
    public void setName(String name) {
        if (name != null && !name.trim().isEmpty()) {
            this.name = name.trim();
        } else {
            throw new IllegalArgumentException("Name cannot be empty!");
        }
    }

    public String getName() {
        return name;
    }

    // Age validation
    public void setAge(int age) {
        if (age >= 0 && age <= 150) {
            this.age = age;
        } else {
            throw new IllegalArgumentException("Age must be between 0 and 150!");
        }
    }

    public int getAge() {
        return age;
    }

    // Email validation (simplified)
    public void setEmail(String email) {
        if (email != null && email.contains("@") && email.contains(".")) {
            this.email = email.toLowerCase();
        } else {
            throw new IllegalArgumentException("Invalid email format!");
        }
    }
```

```
40
41      public String getEmail() {
42          return email;
43      }
44  }
45  public class Main {
47      public static void main(String[] args) {
48          Person person = new Person();
49
50          try {
51              person.setName("Alice");       // Valid
52              person.setAge(25);             // Valid
53              person.setEmail("alice@example.com");  // Valid
54
55              System.out.println("Person created successfully!");
56
57              // These will throw exceptions:
58              // person.setName("");           //
    IllegalArgumentException
59              // person.setAge(200);           //
    IllegalArgumentException
60              // person.setEmail("invalid");   //
    IllegalArgumentException
61
62          } catch (IllegalArgumentException e) {
63              System.out.println("Error: " + e.getMessage());
64          }
65      }
66  }
```

Real-World Example

Student: Can you show a complete real-world example?

Expert: Sure! Let's create a `Product` class for an e-commerce system:

```
1  class Product {
2      // Private fields
3      private String productId;
4      private String name;
5      private double price;
6      private int stockQuantity;
7      private String category;
8
9      // Constructor
```

```java
    public Product(String productId, String name, double price, int stockQuantity, String category) {
        setProductId(productId);
        setName(name);
        setPrice(price);
        setStockQuantity(stockQuantity);
        setCategory(category);
    }

    // Getters and Setters with validation

    public String getProductId() {
        return productId;
    }

    private void setProductId(String productId) {  // Private - cannot change after creation
        if (productId != null && productId.startsWith("PROD-")) {
            this.productId = productId;
        } else {
            throw new IllegalArgumentException("Product ID must start with 'PROD-'");
        }
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        if (name != null && name.length() >= 3) {
            this.name = name;
        } else {
            throw new IllegalArgumentException("Name must be at least 3 characters!");
        }
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        if (price > 0) {
            this.price = price;
        } else {
            throw new IllegalArgumentException("Price must be positive!");
```

```
        }
    }

    public int getStockQuantity() {
        return stockQuantity;
    }

    private void setStockQuantity(int stockQuantity) {  // Private - use specific methods
        if (stockQuantity >= 0) {
            this.stockQuantity = stockQuantity;
        } else {
            throw new IllegalArgumentException("Stock cannot be negative!");
        }
    }

    public String getCategory() {
        return category;
    }

    public void setCategory(String category) {
        this.category = category;
    }

    // Business logic methods

    public boolean isInStock() {
        return stockQuantity > 0;
    }

    public boolean isLowStock() {
        return stockQuantity > 0 && stockQuantity < 10;
    }

    public void addStock(int quantity) {
        if (quantity > 0) {
            stockQuantity += quantity;
            System.out.println("Added " + quantity + " units. New stock: " + stockQuantity);
        }
    }

    public boolean reduceStock(int quantity) {
        if (quantity > 0 && quantity <= stockQuantity) {
            stockQuantity -= quantity;
            System.out.println("Reduced " + quantity + " units. Remaining: " + stockQuantity);
```

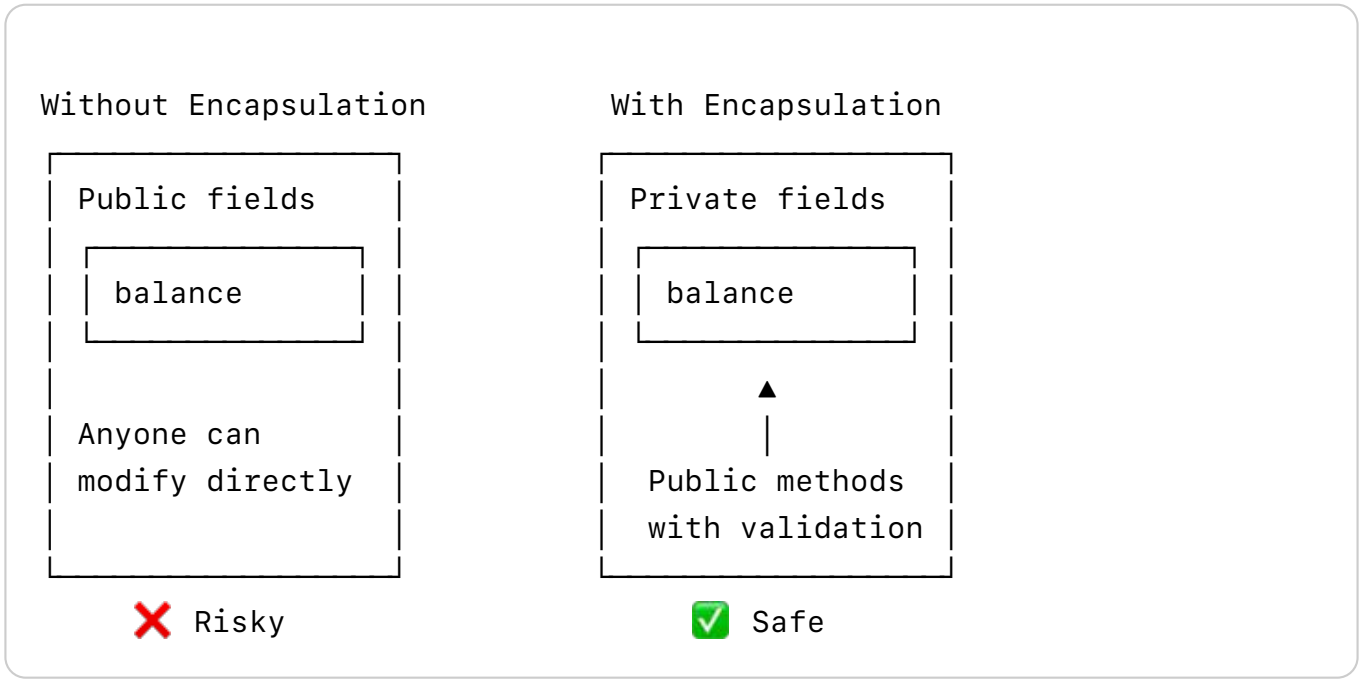
```
            return true;
        } else {
            System.out.println("Cannot reduce stock. Insufficient quantity!");
            return false;
        }
    }

    public void applyDiscount(double percentage) {
        if (percentage > 0 && percentage <= 50) {  // Max 50% discount
            double discount = price * (percentage / 100);
            price -= discount;
            System.out.println("Discount applied! New price: " + price);
        } else {
            System.out.println("Invalid discount percentage!");
        }
    }

    public void displayInfo() {
        System.out.println("=== Product Info ===");
        System.out.println("ID: " + productId);
        System.out.println("Name: " + name);
        System.out.println("Price: $" + price);
        System.out.println("Stock: " + stockQuantity);
        System.out.println("Category: " + category);
        System.out.println("Status: " + (isInStock() ? "Available" : "Out of Stock"));
        if (isLowStock()) {
            System.out.println("WARNING: Low stock!");
        }
    }
}
public class Main {
    public static void main(String[] args) {
        // Create a product
        Product laptop = new Product("PROD-001", "Gaming Laptop", 1200.0, 15, "Electronics");

        laptop.displayInfo();
        System.out.println();

        // Sell some units
        laptop.reduceStock(10);
        System.out.println();

        // Check if low stock
```

```
141         if (laptop.isLowStock()) {
142             System.out.println("Reordering needed!");
143             laptop.addStock(20);
144         }
145         System.out.println();
146 
147         // Apply discount
148         laptop.applyDiscount(10);
149         System.out.println();
150 
151         laptop.displayInfo();
152     }
153 }
```

Output:

```
=== Product Info ===
ID: PROD-001
Name: Gaming Laptop
Price: $1200.0
Stock: 15
Category: Electronics
Status: Available

Reduced 10 units. Remaining: 5

Reordering needed!
Added 20 units. New stock: 25

Discount applied! New price: 1080.0

=== Product Info ===
ID: PROD-001
Name: Gaming Laptop
Price: $1080.0
Stock: 25
Category: Electronics
Status: Available
```

Benefits Visualization



Interview Trap Alert! 🚨

Interviewer might ask: "If we have getters and setters, isn't the data still accessible? What's the point?"

Wrong Answer: "Yes, you're right. Encapsulation doesn't really hide anything."

Correct Answer: "The point isn't just hiding, it's about **controlled access**. With getters and setters, we can: 1. Add validation logic 2. Make fields read-only (getter but no setter) 3. Make fields write-only (setter but no getter) 4. Add logging, security checks, or other logic 5. Change internal implementation without affecting external code"

```java
// Example: Changing internal implementation
class Temperature {
    private double celsius;  // Internal storage in Celsius

    // External interface can be in Fahrenheit
    public void setTemperatureFahrenheit(double fahrenheit) {
        this.celsius = (fahrenheit - 32) * 5/9;  // Convert and store
    }

    public double getTemperatureFahrenheit() {
        return (celsius * 9/5) + 32;  // Convert and return
    }

    // We can change storage to Fahrenheit later without affecting users!
}
```

Chapter Recap

✅ Encapsulation = Data hiding + Controlled access
✅ Make fields private, provide public getters/setters
✅ Add validation logic in setters
✅ Not all fields need both getter and setter
✅ Encapsulation provides flexibility to change implementation

Interview Questions

1. **Q:** What is encapsulation?
   A: Encapsulation is the bundling of data (variables) and methods that operate on that data into a single unit (class), while restricting direct access to some components.

2. **Q:** Why make variables private?
   A: To prevent unauthorized or invalid modifications, maintain data integrity, and provide controlled access through methods.

3. **Q:** What are getters and setters?
   A: Getters (accessors) are methods that return the value of private variables. Setters (mutators) are methods that set the value of private variables, often with validation.

4. **Q:** Can a class be fully encapsulated without setters?

A: Yes! If you only provide getters (no setters), the fields become read-only, which is a valid form of encapsulation.

5. **Q:** What's the benefit of encapsulation?
   A: Data security, validation, flexibility to change implementation, maintainability, and better control over how data is accessed and modified.

Chapter 4: Abstraction

Student–Expert Dialogue

Student: I'm confused about abstraction. Isn't it the same as encapsulation?

Expert: Great question! They're related but different. Let me clarify:

- **Encapsulation** = Hiding data (implementation details)
- **Abstraction** = Hiding complexity (showing only essential features)

Think of a car: - **Abstraction**: You see steering wheel, pedals, gear shift (essential features). You don't see engine internals, transmission details. - **Encapsulation**: The engine parts are sealed in a compartment (data hiding).

Student: Can you give a programming example?

Expert: Sure! Let's start with a real-world scenario:

```java
// WITHOUT Abstraction - User sees too much complexity
class EmailService {
    public void connectToServer(String server) {
        System.out.println("Connecting to " + server);
    }

    public void authenticateUser(String username, String password) {
        System.out.println("Authenticating " + username);
    }

    public void prepareEmailHeaders(String from, String to) {
        System.out.println("Preparing headers from " + from + " to " + to);
    }

    public void encryptMessage(String message) {
        System.out.println("Encrypting message");
    }

    public void transmitData(String data) {
        System.out.println("Transmitting data");
    }

    public void closeConnection() {
        System.out.println("Closing connection");
    }
}
public class Main {
    public static void main(String[] args) {
        EmailService emailService = new EmailService();

        // User has to know ALL these steps - TOO COMPLEX!
        emailService.connectToServer("smtp.gmail.com");
        emailService.authenticateUser("user@gmail.com", "password");
        emailService.prepareEmailHeaders("user@gmail.com", "friend@gmail.com");
        emailService.encryptMessage("Hello!");
        emailService.transmitData("Hello!");
        emailService.closeConnection();
    }
}
```

Student: That does look complicated!

Expert: Exactly! Now let's apply abstraction:

```java
// WITH Abstraction - User sees only what's necessary
class EmailService {
    // Private methods - hidden complexity
    private void connectToServer(String server) {
        System.out.println("Connecting to " + server);
    }

    private void authenticateUser(String username, String password) {
        System.out.println("Authenticating " + username);
    }

    private void prepareEmailHeaders(String from, String to) {
        System.out.println("Preparing headers");
    }

    private void encryptMessage(String message) {
        System.out.println("Encrypting message");
    }

    private void transmitData(String data) {
        System.out.println("Transmitting data");
    }

    private void closeConnection() {
        System.out.println("Closing connection");
    }

    // Public method - simple interface (ABSTRACTION!)
    public void sendEmail(String from, String to, String message, String password) {
        System.out.println("=== Sending Email ===");
        connectToServer("smtp.gmail.com");
        authenticateUser(from, password);
        prepareEmailHeaders(from, to);
        encryptMessage(message);
        transmitData(message);
        closeConnection();
        System.out.println("Email sent successfully!");
    }
}
public class Main {
    public static void main(String[] args) {
        EmailService emailService = new EmailService();

```

```
45         // Simple! User doesn't need to know internal complexity
46         emailService.sendEmail("user@gmail.com",
   "friend@gmail.com", "Hello!", "password");
47     }
48 }
```

Output:

```
=== Sending Email ===
Connecting to smtp.gmail.com
Authenticating user@gmail.com
Preparing headers
Encrypting message
Transmitting data
Closing connection
Email sent successfully!
```

Student: So abstraction is about providing a simple interface?

Expert: Exactly! Let me show you the concept visually:

```
Abstraction Layers:

User View (Simple Interface)
┌──────────────────────────────┐
│   sendEmail()                │  ← What user sees
└──────────────────────────────┘
            │
            ▼
Hidden Complexity (Implementation)
┌──────────────────────────────┐
│ connectToServer()            │
│ authenticateUser()           │
│ prepareEmailHeaders()        │  ← What user doesn't see
│ encryptMessage()             │
│ transmitData()               │
│ closeConnection()            │
└──────────────────────────────┘
```

Student: How is this different from encapsulation again?

Expert: Let me show you side by side:

```java
class BankAccount {
    // ENCAPSULATION - Hiding data
    private double balance;  // Hidden data

    public double getBalance() {  // Controlled access
        return balance;
    }

    public void setBalance(double balance) {  // Controlled modification
        if (balance >= 0) {
            this.balance = balance;
        }
    }

    // ABSTRACTION - Hiding complexity
    private void validateAccount() {
        System.out.println("Validating account...");
    }

    private void checkFraudDetection() {
        System.out.println("Checking for fraud...");
    }

    private void updateTransactionLog() {
        System.out.println("Updating logs...");
    }

    private void notifyUser() {
        System.out.println("Sending notification...");
    }

    // Simple public interface (ABSTRACTION)
    public void withdraw(double amount) {
        System.out.println("=== Processing Withdrawal ===");
        validateAccount();
        checkFraudDetection();

        if (amount > 0 && amount <= balance) {
            balance -= amount;  // Using encapsulated data
            updateTransactionLog();
            notifyUser();
            System.out.println("Withdrawal successful: $" +
```

```
    amount);
43          } else {
44              System.out.println("Invalid amount or insufficient
    balance!");
45          }
46      }
47  }
48  public class Main {
50      public static void main(String[] args) {
51          BankAccount account = new BankAccount();
52          account.setBalance(1000);
53
54          // User calls one simple method
55          // All complexity is hidden
56          account.withdraw(200);
57      }
58  }
```

Output:

```
=== Processing Withdrawal ===
Validating account...
Checking for fraud...
Updating logs...
Sending notification...
Withdrawal successful: $200.0
```

Abstract Classes

Student: I've heard about "abstract classes." Are they related to abstraction?

Expert: Yes! Abstract classes are a way to implement abstraction in Java. Let me explain:

```
1  // Abstract class - cannot be instantiated directly
2  abstract class Animal {
3      // Regular field
4      protected String name;
5
6      // Constructor
7      public Animal(String name) {
8          this.name = name;
9      }
```

```java

    // Concrete method (has implementation)
    public void sleep() {
        System.out.println(name + " is sleeping...");
    }

    // Abstract method (no implementation) - MUST be implemented by subclass
    public abstract void makeSound();

    // Abstract method
    public abstract void move();
}
// Concrete class - must implement all abstract methods
class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }

    @Override
    public void makeSound() {
        System.out.println(name + " says: Woof! Woof!");
    }

    @Override
    public void move() {
        System.out.println(name + " runs on four legs");
    }
}
class Bird extends Animal {
    public Bird(String name) {
        super(name);
    }

    @Override
    public void makeSound() {
        System.out.println(name + " says: Chirp! Chirp!");
    }

    @Override
    public void move() {
        System.out.println(name + " flies in the sky");
    }
}
public class Main {
    public static void main(String[] args) {
        // Cannot do this:
        // Animal animal = new Animal("Generic");  // Compile
```

```
    error!
60
61          // Must create concrete subclass objects
62          Animal dog = new Dog("Buddy");
63          Animal bird = new Bird("Tweety");
64
65          dog.makeSound();    // Woof! Woof!
66          dog.move();         // runs on four legs
67          dog.sleep();        // is sleeping...
68
69          System.out.println();
70
71          bird.makeSound();   // Chirp! Chirp!
72          bird.move();        // flies in the sky
73          bird.sleep();       // is sleeping...
74      }
75  }
```

Line-by-line explanation: - `abstract class Animal` - Cannot create objects directly - `public abstract void makeSound()` - No implementation, subclasses MUST provide it - `public void sleep()` - Regular method, has implementation, inherited by subclasses - `class Dog extends Animal` - Concrete class, must implement all abstract methods - `@Override` - Indicates we're implementing the abstract method

Student: When should I use abstract classes?

Expert: Use abstract classes when: 1. You want to share code among related classes 2. You want to define a template that subclasses must follow 3. You have some common behavior (concrete methods) and some that varies (abstract methods)

Let me show you a practical example:

```
1  // Abstract class for payment processing
2  abstract class PaymentProcessor {
3      protected String merchantId;
4      protected double amount;
5
6      public PaymentProcessor(String merchantId, double amount) {
7          this.merchantId = merchantId;
8          this.amount = amount;
9      }
10
11     // Template method - defines the process
12     public final void processPayment() {
13         System.out.println("=== Processing Payment ===");
```

```
14          validatePayment();
15
16          if (connectToPaymentGateway()) {
17              if (authorizePayment()) {
18                  capturePayment();
19                  sendConfirmation();
20                  System.out.println("Payment successful!");
21              } else {
22                  System.out.println("Payment authorization
    failed!");
23              }
24          } else {
25              System.out.println("Cannot connect to payment
    gateway!");
26          }
27      }
28
29      // Common implementation for all payment types
30      private void validatePayment() {
31          System.out.println("Validating payment amount: $" +
    amount);
32          if (amount <= 0) {
33              throw new IllegalArgumentException("Invalid
    amount!");
34          }
35      }
36
37      // Common implementation
38      private void sendConfirmation() {
39          System.out.println("Sending confirmation email...");
40      }
41
42      // Abstract methods - each payment type implements
    differently
43      protected abstract boolean connectToPaymentGateway();
44      protected abstract boolean authorizePayment();
45      protected abstract void capturePayment();
46  }
47  class CreditCardPayment extends PaymentProcessor {
48      private String cardNumber;
49
50      public CreditCardPayment(String merchantId, double amount,
    String cardNumber) {
51          super(merchantId, amount);
52          this.cardNumber = cardNumber;
53      }
54
55      @Override
56
```

```
    protected boolean connectToPaymentGateway() {
        System.out.println("Connecting to Credit Card Gateway...");
        return true;
    }

    @Override
    protected boolean authorizePayment() {
        System.out.println("Authorizing credit card: " + maskCardNumber());
        return true;
    }

    @Override
    protected void capturePayment() {
        System.out.println("Capturing payment from credit card");
    }

    private String maskCardNumber() {
        return "****-****-****-" + cardNumber.substring(cardNumber.length() - 4);
    }
}
class PayPalPayment extends PaymentProcessor {
    private String email;

    public PayPalPayment(String merchantId, double amount, String email) {
        super(merchantId, amount);
        this.email = email;
    }

    @Override
    protected boolean connectToPaymentGateway() {
        System.out.println("Connecting to PayPal Gateway...");
        return true;
    }

    @Override
    protected boolean authorizePayment() {
        System.out.println("Authorizing PayPal account: " + email);
        return true;
    }

    @Override
    protected void capturePayment() {
        System.out.println("Capturing payment from PayPal");
```

```
101     }
102 }
103 class BitcoinPayment extends PaymentProcessor {
104     private String walletAddress;
105 
106     public BitcoinPayment(String merchantId, double amount,
    String walletAddress) {
107         super(merchantId, amount);
108         this.walletAddress = walletAddress;
109     }
110 
111     @Override
112     protected boolean connectToPaymentGateway() {
113         System.out.println("Connecting to Bitcoin Network...");
114         return true;
115     }
116 
117     @Override
118     protected boolean authorizePayment() {
119         System.out.println("Authorizing Bitcoin wallet: " +
    walletAddress);
120         return true;
121     }
122 
123     @Override
124     protected void capturePayment() {
125         System.out.println("Capturing payment from Bitcoin
    wallet");
126     }
127 }
128 public class Main {
129     public static void main(String[] args) {
130         PaymentProcessor creditCard = new
    CreditCardPayment("M123", 100.0, "1234567890123456");
131         creditCard.processPayment();
132 
133         System.out.println("\n");
134 
135         PaymentProcessor paypal = new PayPalPayment("M123",
    200.0, "user@example.com");
136         paypal.processPayment();
137 
138         System.out.println("\n");
139 
140         PaymentProcessor bitcoin = new BitcoinPayment("M123",
    300.0, "1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa");
141         bitcoin.processPayment();
142     }
```

```
145 }
```

Output:

```
=== Processing Payment ===
Validating payment amount: $100.0
Connecting to Credit Card Gateway...
Authorizing credit card: ****-****-****-3456
Capturing payment from credit card
Sending confirmation email...
Payment successful!


=== Processing Payment ===
Validating payment amount: $200.0
Connecting to PayPal Gateway...
Authorizing PayPal account: user@example.com
Capturing payment from PayPal
Sending confirmation email...
Payment successful!


=== Processing Payment ===
Validating payment amount: $300.0
Connecting to Bitcoin Network...
Authorizing Bitcoin wallet: 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa
Capturing payment from Bitcoin wallet
Sending confirmation email...
Payment successful!
```

Student: That’s powerful! The overall process is the same, but each payment type has its own implementation.

Expert: Exactly! This is called the **Template Method Pattern**. The abstract class defines the skeleton, and subclasses fill in the details.

Key Rules for Abstract Classes

Student: What are the rules I need to remember?

Expert: Here are the key rules:

```java
// Rule demonstrations
abstract class AbstractRules {

    // ✅ Rule 1: Can have both abstract and concrete methods
    public abstract void abstractMethod();

    public void concreteMethod() {
        System.out.println("This has implementation");
    }

    // ✅ Rule 2: Can have constructors
    public AbstractRules() {
        System.out.println("Abstract class constructor");
    }

    // ✅ Rule 3: Can have instance variables
    private int value;

    // ✅ Rule 4: Can have static methods
    public static void staticMethod() {
        System.out.println("Static method in abstract class");
    }

    // ✅ Rule 5: Can have final methods (cannot be overridden)
    public final void finalMethod() {
        System.out.println("Cannot override this");
    }

    // ✅ Rule 6: Abstract methods cannot be private
    // private abstract void invalid();  // Compile error!

    // ✅ Rule 7: Abstract methods cannot be final
    // public final abstract void invalid();  // Compile error!

    // ✅ Rule 8: Abstract methods cannot be static
    // public static abstract void invalid();  // Compile error!
}
// ✅ Rule 9: Concrete subclass must implement ALL abstract methods
class ConcreteClass extends AbstractRules {
    @Override
    public void abstractMethod() {
        System.out.println("Implemented!");
    }
}
// ✅ Rule 10: Abstract subclass doesn't need to implement abstract methods
```

```java
abstract class AnotherAbstractClass extends AbstractRules {
    // Can add more abstract methods
    public abstract void anotherMethod();
}
public class Main {
    public static void main(String[] args) {
        // ❌ Rule 11: Cannot instantiate abstract class
        // AbstractRules obj = new AbstractRules();  // Compile error!

        // ✅ But can create reference of abstract type
        AbstractRules obj = new ConcreteClass();
        obj.abstractMethod();
        obj.concreteMethod();
        obj.finalMethod();

        // ✅ Can call static methods using class name
        AbstractRules.staticMethod();
    }
}
```

Abstraction Levels

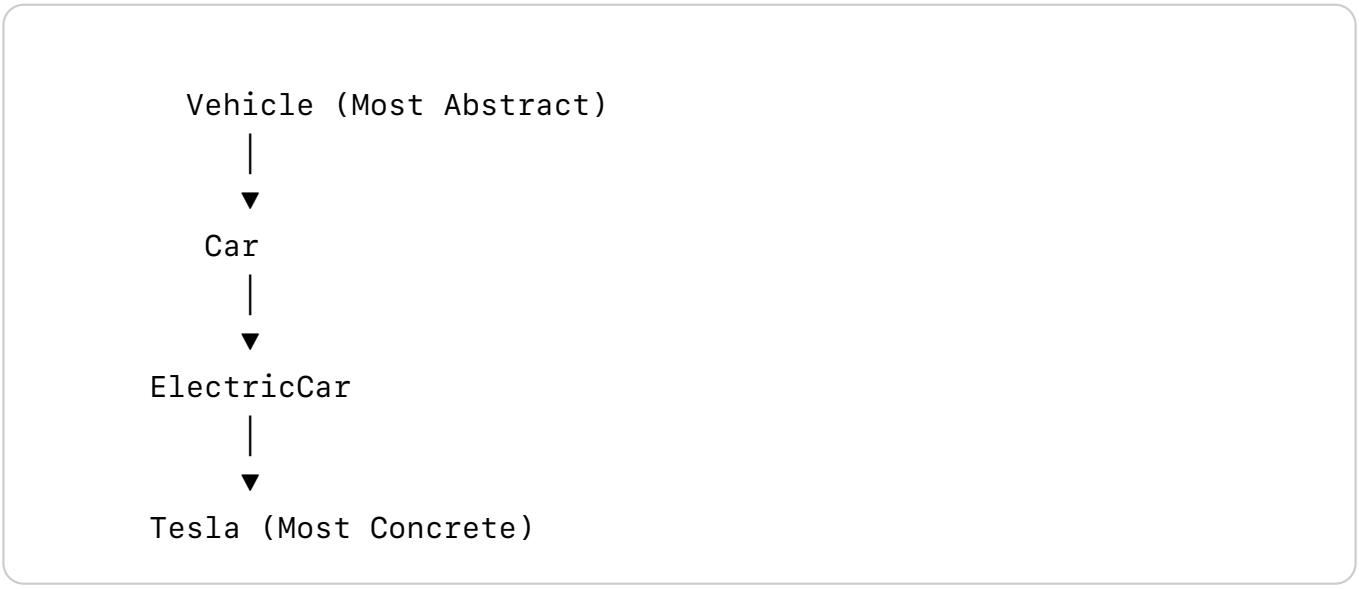
Student: Can you show me different levels of abstraction?

Expert: Great question! Let me demonstrate:

```java
// High-level abstraction
abstract class Vehicle {
    public abstract void start();
    public abstract void stop();
}
// Medium-level abstraction
abstract class Car extends Vehicle {
    protected int numberOfDoors;

    public abstract void openTrunk();

    // Partially implemented
    @Override
    public void start() {
        System.out.println("Turning key...");
        startEngine();  // Abstract - subclass decides
    }

```

```
    protected abstract void startEngine();
}
// Low-level abstraction (more specific)
abstract class ElectricCar extends Car {
    protected int batteryCapacity;

    @Override
    protected void startEngine() {
        System.out.println("Powering electric motor...");
    }

    public abstract void chargeBattery();
}
// Concrete implementation (no abstraction)
class Tesla extends ElectricCar {
    public Tesla() {
        this.numberOfDoors = 4;
        this.batteryCapacity = 100;
    }

    @Override
    public void stop() {
        System.out.println("Regenerative braking engaged");
    }

    @Override
    public void openTrunk() {
        System.out.println("Trunk opening automatically");
    }

    @Override
    public void chargeBattery() {
        System.out.println("Supercharging at 250kW");
    }
}
public class Main {
    public static void main(String[] args) {
        Tesla myTesla = new Tesla();

        myTesla.start();           // From Car
        myTesla.openTrunk();       // From Tesla
        myTesla.chargeBattery();   // From Tesla
        myTesla.stop();            // From Tesla
    }
}
```

Abstraction Hierarchy:



Interview Trap Alert! 🚨

Interviewer might ask: “Can an abstract class have a constructor? If yes, what’s the purpose?”

Wrong Answer: “No, abstract classes cannot have constructors because we can’t create objects of abstract classes.”

Correct Answer: “Yes, abstract classes CAN have constructors. Even though we can’t instantiate an abstract class directly, the constructor is called when a subclass object is created. It’s used to initialize common fields of the abstract class.”

```java
abstract class Employee {
    protected String name;
    protected String id;

    // Constructor in abstract class
    public Employee(String name, String id) {
        this.name = name;
        this.id = id;
        System.out.println("Employee constructor called");
    }

    public abstract double calculateSalary();
}
class FullTimeEmployee extends Employee {
    private double monthlySalary;

    public FullTimeEmployee(String name, String id, double monthlySalary) {
        super(name, id);  // Calls abstract class constructor
        this.monthlySalary = monthlySalary;
        System.out.println("FullTimeEmployee constructor called");
    }

    @Override
    public double calculateSalary() {
        return monthlySalary;
    }
}
public class Main {
    public static void main(String[] args) {
        Employee emp = new FullTimeEmployee("John", "E001", 5000);
        // Output:
        // Employee constructor called
        // FullTimeEmployee constructor called
    }
}
```

Chapter Recap

✅ Abstraction = Hiding complexity, showing only essential features
✅ Different from encapsulation (which hides data)
✅ Abstract classes provide partial implementation

✅ Abstract methods must be implemented by subclasses
✅ Abstract classes can have constructors, fields, and concrete methods
✅ Cannot instantiate abstract classes directly

Interview Questions

1. **Q:** What is abstraction?
   A: Abstraction is the process of hiding implementation details and showing only essential features/functionality to the user.

2. **Q:** What is an abstract class?
   A: A class declared with the `abstract` keyword that cannot be instantiated and may contain abstract methods (without implementation) that must be implemented by subclasses.

3. **Q:** Can an abstract class have concrete methods?
   A: Yes, abstract classes can have both abstract methods (without implementation) and concrete methods (with implementation).

4. **Q:** What's the difference between abstraction and encapsulation?
   A: Abstraction hides complexity (implementation details), while encapsulation hides data (using private variables and public methods).

5. **Q:** Can we create a constructor in an abstract class?
   A: Yes, abstract classes can have constructors. They are called when a subclass object is created.

6. **Q:** Can an abstract method be private?
   A: No, abstract methods cannot be private because they must be overridden by subclasses.

Chapter 5: Inheritance

Student–Expert Dialogue

Student: I've heard that inheritance is one of the most important OOP concepts. What exactly is it?

Expert: Inheritance is a mechanism where one class acquires the properties and behaviors of another class. Think of it like family inheritance—children inherit traits from parents.

```
Real-world analogy:

        Animal (Parent)
        - breathe()
        - eat()
            |
            ├──────────┬──────────┐
            ▼          ▼          ▼
         Dog        Cat        Bird
      - bark()   - meow()   - fly()
```

Student: Can you show me a code example?

Expert: Absolutely! Let's start with a basic example:

```java
// Parent class (also called superclass or base class)
class Animal {
    String name;
    int age;

    public void eat() {
        System.out.println(name + " is eating...");
    }

    public void sleep() {
        System.out.println(name + " is sleeping...");
    }
}
// Child class (also called subclass or derived class)
class Dog extends Animal {
    String breed;

    public void bark() {
        System.out.println(name + " says: Woof! Woof!");
    }

    public void wagTail() {
        System.out.println(name + " is wagging tail!");
    }
}
class Cat extends Animal {
    String color;

```

```
    public void meow() {
        System.out.println(name + " says: Meow! Meow!");
    }

    public void scratch() {
        System.out.println(name + " is scratching!");
    }
}
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.name = "Buddy";       // Inherited from Animal
        dog.age = 3;              // Inherited from Animal
        dog.breed = "Labrador";   // Dog's own property

        dog.eat();       // Inherited method
        dog.sleep();     // Inherited method
        dog.bark();      // Dog's own method
        dog.wagTail();   // Dog's own method

        System.out.println();

        Cat cat = new Cat();
        cat.name = "Whiskers";    // Inherited from Animal
        cat.age = 2;              // Inherited from Animal
        cat.color = "Orange";     // Cat's own property

        cat.eat();       // Inherited method
        cat.sleep();     // Inherited method
        cat.meow();      // Cat's own method
        cat.scratch();   // Cat's own method
    }
}
```

Output:

```
Buddy is eating...
Buddy is sleeping...
Buddy says: Woof! Woof!
Buddy is wagging tail!

Whiskers is eating...
Whiskers is sleeping...
Whiskers says: Meow! Meow!
Whiskers is scratching!
```

Line-by-line explanation: - `class Dog extends Animal` - Dog inherits from Animal - `dog.name = "Buddy"` - Can access inherited field - `dog.eat()` - Can call inherited method - `dog.bark()` - Can call its own method

Student: So the child class gets everything from the parent?

Expert: Almost! Let me show you what's inherited and what's not:

```java
class Parent {
    // ✅ Inherited by child
    public int publicField = 1;
    protected int protectedField = 2;

    // ❌ NOT inherited by child
    private int privateField = 3;

    // ✅ Inherited
    public void publicMethod() {
        System.out.println("Public method");
    }

    // ✅ Inherited
    protected void protectedMethod() {
        System.out.println("Protected method");
    }

    // ❌ NOT inherited
    private void privateMethod() {
        System.out.println("Private method");
    }
}
class Child extends Parent {
    public void testAccess() {
        System.out.println(publicField);      // ✅ OK
        System.out.println(protectedField);   // ✅ OK
        // System.out.println(privateField);  // ❌ Compile error!

        publicMethod();      // ✅ OK
        protectedMethod();   // ✅ OK
        // privateMethod();  // ❌ Compile error!
    }
}
public class Main {
    public static void main(String[] args) {
        Child child = new Child();
        child.testAccess();
    }
}
```

Inheritance Access Rules:

Modifier	Class	Subclass	Outside
public	✅	✅	✅
protected	✅	✅	❌
default	✅	✅ (pkg)	❌
private	✅	❌	❌

Types of Inheritance

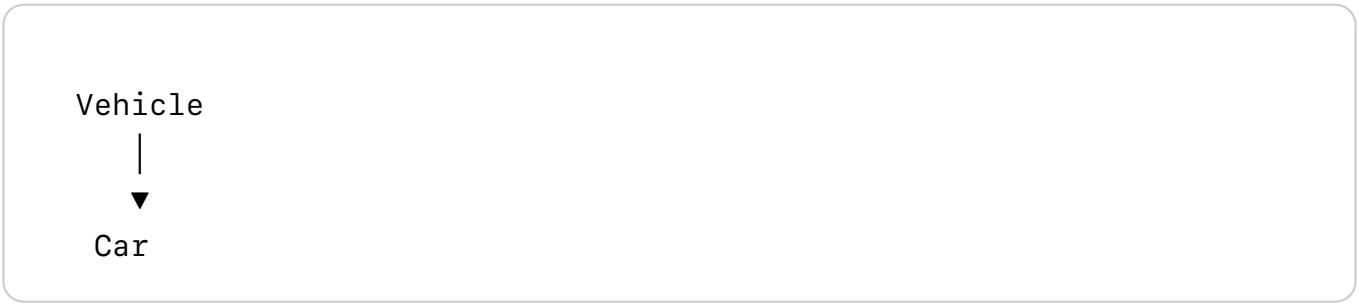
Student: Are there different types of inheritance?

Expert: Yes! Let me show you:

1. Single Inheritance

```java
// One parent, one child
class Vehicle {
    void start() {
        System.out.println("Vehicle starting...");
    }
}
class Car extends Vehicle {
    void drive() {
        System.out.println("Car driving...");
    }
}
public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        car.start();  // From Vehicle
        car.drive();  // From Car
    }
}
```

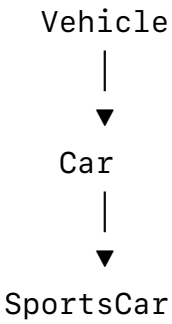
Diagram:



2. Multilevel Inheritance

```java
// Chain of inheritance
class Vehicle {
    void start() {
        System.out.println("Vehicle starting...");
    }
}
class Car extends Vehicle {
    void drive() {
        System.out.println("Car driving...");
    }
}
class SportsCar extends Car {
    void turboBoost() {
        System.out.println("Turbo boost activated!");
    }
}
public class Main {
    public static void main(String[] args) {
        SportsCar sportsCar = new SportsCar();
        sportsCar.start();       // From Vehicle
        sportsCar.drive();       // From Car
        sportsCar.turboBoost();  // From SportsCar
    }
}
```

Diagram:

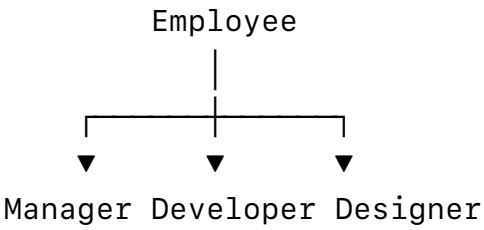


3. Hierarchical Inheritance

```java
// Multiple children from one parent
class Employee {
    String name;
    double baseSalary;

    void work() {
        System.out.println(name + " is working...");
    }
}
class Manager extends Employee {
    int teamSize;

    void conductMeeting() {
        System.out.println(name + " is conducting a meeting");
    }
}
class Developer extends Employee {
    String programmingLanguage;

    void writeCode() {
        System.out.println(name + " is writing " +
programmingLanguage + " code");
    }
}
class Designer extends Employee {
    String designTool;

    void createDesign() {
        System.out.println(name + " is designing with " +
designTool);
    }
}
public class Main {
```

```
36      public static void main(String[] args) {
37          Manager mgr = new Manager();
38          mgr.name = "Alice";
39          mgr.work();
40          mgr.conductMeeting();
41
42          Developer dev = new Developer();
43          dev.name = "Bob";
44          dev.programmingLanguage = "Java";
45          dev.work();
46          dev.writeCode();
47
48          Designer designer = new Designer();
49          designer.name = "Charlie";
50          designer.designTool = "Figma";
51          designer.work();
52          designer.createDesign();
53      }
54  }
```

Diagram:



4. Multiple Inheritance (NOT supported in Java with classes!)

```java
// ❌ This is NOT allowed in Java!
/*
class Father {
    void property() {
        System.out.println("Father's property");
    }
}
class Mother {
    void property() {
        System.out.println("Mother's property");
    }
}
// This causes "Diamond Problem"
class Child extends Father, Mother {  // ❌ Compile error!
    // Which property() method should Child inherit?
}
*/
```

Student: Why doesn't Java support multiple inheritance?

Expert: Because of the **Diamond Problem**:

```
Diamond Problem:

      A
     / \
    B   C
     \ /
      D

If B and C both override a method from A,
which version should D inherit?
```

However, Java supports multiple inheritance through **interfaces** (we'll cover this later).

Constructors in Inheritance

Student: What happens to constructors when we use inheritance?

Expert: Great question! Constructors are NOT inherited, but they ARE called. Let me show you:

```java
class Parent {
    String parentName;

    // Parent constructor
    public Parent() {
        System.out.println("Parent constructor called");
        parentName = "Parent";
    }

    public Parent(String name) {
        System.out.println("Parent parameterized constructor called");
        this.parentName = name;
    }
}
class Child extends Parent {
    String childName;

    // Child constructor
    public Child() {
        // super() is automatically called here (invisible)
        System.out.println("Child constructor called");
        childName = "Child";
    }

    public Child(String parentName, String childName) {
        super(parentName);  // Explicitly calling parent constructor
        System.out.println("Child parameterized constructor called");
        this.childName = childName;
    }
}
public class Main
    {
    public static void main(String[] args) {
        System.out.println("=== Creating Child with default constructor ===");
        Child child1 = new Child();

        System.out.println("\n=== Creating Child with parameterized constructor ===");
        Child child2 = new Child("SuperParent", "SuperChild");

        System.out.println("\nParent name: " + child2.parentName);
        System.out.println("Child name: " + child2.childName);
```

```
    }
}
```

Output:

```
=== Creating Child with default constructor ===
Parent constructor called
Child constructor called

=== Creating Child with parameterized constructor ===
Parent parameterized constructor called
Child parameterized constructor called

Parent name: SuperParent
Child name: SuperChild
```

Student: So the parent constructor is always called first?

Expert: Exactly! Here's the order:

```
Constructor Call Order:
┌──────────────────────────────────┐
│ 1. Parent constructor            │
│    ↓                             │
│ 2. Child constructor             │
└──────────────────────────────────┘

If multilevel:
┌──────────────────────────────────┐
│ 1. Grandparent constructor       │
│    ↓                             │
│ 2. Parent constructor            │
│    ↓                             │
│ 3. Child constructor             │
└──────────────────────────────────┘
```

Student: What if I don't call `super()`?

Expert: Java automatically inserts `super()` (no-argument) as the first line of your

constructor:

```java
class Parent {
    public Parent() {
        System.out.println("Parent no-arg constructor");
    }

    public Parent(String name) {
        System.out.println("Parent parameterized constructor: " + name);
    }
}
class Child extends Parent {
    // This constructor:
    public Child() {
        System.out.println("Child constructor");
    }

    // Is automatically converted to:
    /*
    public Child() {
        super();  // Automatically added by Java
        System.out.println("Child constructor");
    }
    */
}
// ❌ This will cause an error:
class ProblematicChild extends Parent {
    public ProblematicChild(String name) {
        // Java tries to call super() automatically
        // But Parent doesn't have a no-arg constructor anymore!
        System.out.println("Child: " + name);
    }
}
```

To fix the error:

```java
class FixedChild extends Parent {
    public FixedChild(String name) {
        super(name);  // Explicitly call parameterized parent constructor
        System.out.println("Child: " + name);
    }
}
```

Method Overriding in Inheritance

Student: Can a child class change the behavior of inherited methods?

Expert: Yes! That's called **method overriding**. Let me show you:

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

    void eat() {
        System.out.println("Animal is eating");
    }
}
class Dog extends Animal {
    // Overriding the makeSound method
    @Override
    void makeSound() {
        System.out.println("Dog says: Woof! Woof!");
    }

    // Inherited eat() method is not overridden
}
class Cat extends Animal {
    // Overriding the makeSound method
    @Override
    void makeSound() {
        System.out.println("Cat says: Meow! Meow!");
    }

    // Overriding the eat method too
    @Override
    void eat() {
        System.out.println("Cat is eating fish");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal();
        animal.makeSound();  // Animal makes a sound
        animal.eat();        // Animal is eating

        System.out.println();

        Dog dog = new Dog();
```

```
        dog.makeSound();  // Dog says: Woof! Woof! (Overridden)
        dog.eat();        // Animal is eating (Inherited)

        System.out.println();

        Cat cat = new Cat();
        cat.makeSound();  // Cat says: Meow! Meow! (Overridden)
        cat.eat();        // Cat is eating fish (Overridden)
    }
}
```

Student: What's the `@Override` annotation for?

Expert: The `@Override` annotation is optional but highly recommended. It tells the compiler "I'm intentionally overriding a parent method." If you make a mistake, the compiler will catch it:

```
class Parent {
    void display() {
        System.out.println("Parent display");
    }
}
class Child extends Parent {
    // ✅ Correct override
    @Override
    void display() {
        System.out.println("Child display");
    }

    // ❌ Typo - compiler will catch this because of @Override
    /*
    @Override
    void displya() {  // Compile error: method doesn't override anything
        System.out.println("Child display");
    }
    */
}
```

Rules for Method Overriding

Expert: Here are the important rules:

```
class Parent {
```

```java
    // Original method
    protected String getMessage(int value) {
        return "Parent: " + value;
    }

    public void display() {
        System.out.println("Parent display");
    }

    public final void cannotOverride() {
        System.out.println("This cannot be overridden");
    }
}
class Child extends Parent {
    // ✅ Rule 1: Same method signature (name + parameters)
    @Override
    protected String getMessage(int value) {
        return "Child: " + value;
    }

    // ✅ Rule 2: Return type must be same or covariant (subtype)
    @Override
    public void display() {
        System.out.println("Child display");
    }

    // ✅ Rule 3: Access modifier can be same or more accessible
    // protected -> public is OK
    // public -> protected is NOT OK

    // ❌ Rule 4: Cannot override final methods
    /*
    @Override
    public void cannotOverride() {  // Compile error!
        System.out.println("Trying to override");
    }
    */

    // ❌ Rule 5: Cannot reduce access level
    /*
    @Override
    private String getMessage(int value) {  // Compile error!
        return "Child: " + value;
    }
    */
}
```

Overriding Rules Summary:

```
┌──────────────────────────────────────────────┐
│ ✅ Must have same method name                │
│ ✅ Must have same parameters                 │
│ ✅ Must have same or covariant return type   │
│ ✅ Can have same or more accessible modifier │
│ ❌ Cannot override final methods             │
│ ❌ Cannot override static methods (hiding)   │
│ ❌ Cannot reduce access level                │
└──────────────────────────────────────────────┘
```

Real-World Example: Employee Management System

Student: Can you show a complete real-world example?

Expert: Sure! Let's build an employee management system:

```java
// Base class
class Employee {
    protected String name;
    protected String employeeId;
    protected double baseSalary;

    public Employee(String name, String employeeId, double baseSalary) {
        this.name = name;
        this.employeeId = employeeId;
        this.baseSalary = baseSalary;
    }

    // Method to be overridden
    public double calculateSalary() {
        return baseSalary;
    }

    // Method to be overridden
    public void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("ID: " + employeeId);
        System.out.println("Salary: $" + calculateSalary());
    }
```

```
24
25      // Common method (not overridden)
26      public void clockIn() {
27          System.out.println(name + " clocked in at " +
    java.time.LocalTime.now());
28      }
29  }
30  // Full-time employee
31  class FullTimeEmployee extends Employee {
32      private double bonus;
33      private double benefits;
34
35      public FullTimeEmployee(String name, String employeeId,
    double baseSalary, double bonus, double benefits) {
36          super(name, employeeId, baseSalary);
37          this.bonus = bonus;
38          this.benefits = benefits;
39      }
40
41      @Override
42      public double calculateSalary() {
43          return baseSalary + bonus + benefits;
44      }
45
46      @Override
47      public void displayInfo() {
48          System.out.println("=== Full-Time Employee ===");
49          super.displayInfo();  // Call parent's displayInfo
50          System.out.println("Bonus: $" + bonus);
51          System.out.println("Benefits: $" + benefits);
52      }
53  }
54  // Part-time employee
55  class PartTimeEmployee extends Employee {
56      private int hoursWorked;
57      private double hourlyRate;
58
59      public PartTimeEmployee(String name, String employeeId, int
    hoursWorked, double hourlyRate) {
60          super(name, employeeId, 0);  // No base salary
61          this.hoursWorked = hoursWorked;
62          this.hourlyRate = hourlyRate;
63      }
64
65      @Override
66      public double calculateSalary() {
67          return hoursWorked * hourlyRate;
68      }
69
70
```

```
71
72      @Override
73      public void displayInfo() {
74          System.out.println("=== Part-Time Employee ===");
75          System.out.println("Name: " + name);
76          System.out.println("ID: " + employeeId);
77          System.out.println("Hours Worked: " + hoursWorked);
78          System.out.println("Hourly Rate: $" + hourlyRate);
79          System.out.println("Total Salary: $" +
    calculateSalary());
80      }
81  }
82  // Contractor
84  class Contractor extends Employee {
85      private int projectsCompleted;
86      private double ratePerProject;
87
88      public Contractor(String name, String employeeId, int
    projectsCompleted, double ratePerProject) {
89          super(name, employeeId, 0);
90          this.projectsCompleted = projectsCompleted;
91          this.ratePerProject = ratePerProject;
92      }
93
94      @Override
95      public double calculateSalary() {
96          return projectsCompleted * ratePerProject;
97      }
98
99      @Override
100     public void displayInfo() {
101         System.out.println("=== Contractor ===");
102         System.out.println("Name: " + name);
103         System.out.println("ID: " + employeeId);
104         System.out.println("Projects Completed: " +
    projectsCompleted);
105         System.out.println("Rate per Project: $" +
    ratePerProject);
106         System.out.println("Total Payment: $" +
    calculateSalary());
107     }
108 }
110 public class Main {
111     public static void main(String[] args) {
112         // Create different types of employees
113         FullTimeEmployee fte = new FullTimeEmployee("Alice",
    "FT001", 5000, 1000, 500);
114         PartTimeEmployee pte = new PartTimeEmployee("Bob",
```

```
    "PT001", 80, 25);
115         Contractor contractor = new Contractor("Charlie",
    "CT001", 5, 2000);
116
117         // Display information
118         fte.displayInfo();
119         System.out.println();
120
121         pte.displayInfo();
122         System.out.println();
123
124         contractor.displayInfo();
125         System.out.println();
126
127         // Common functionality
128         fte.clockIn();
129         pte.clockIn();
130         contractor.clockIn();
131
132         System.out.println();
133
134         // Polymorphism - treating all as Employee
135         Employee[] employees = {fte, pte, contractor};
136
137         double totalPayroll = 0;
138         System.out.println("=== Payroll Summary ===");
139         for (Employee emp : employees) {
140             double salary = emp.calculateSalary();
141             totalPayroll += salary;
142             System.out.println(emp.name + ": $" + salary);
143         }
144         System.out.println("Total Payroll: $" + totalPayroll);
145     }
146 }
```

Output:

```
=== Full-Time Employee ===
Name: Alice
ID: FT001
Salary: $6500.0
Bonus: $1000.0
Benefits: $500.0

=== Part-Time Employee ===
Name: Bob
ID: PT001
Hours Worked: 80
Hourly Rate: $25.0
Total Salary: $2000.0

=== Contractor ===
Name: Charlie
ID: CT001
Projects Completed: 5
Rate per Project: $2000.0
Total Payment: $10000.0

Alice clocked in at 10:30:45.123
Bob clocked in at 10:30:45.124
Charlie clocked in at 10:30:45.125

=== Payroll Summary ===
Alice: $6500.0
Bob: $2000.0
Charlie: $10000.0
Total Payroll: $18500.0
```

Using super Keyword

Student: I saw `super.displayInfo()` in the code. What does `super` do?

Expert: The `super` keyword refers to the parent class. It has three main uses:

```java
class Parent {
    String name = "Parent";

```

```
    public Parent() {
        System.out.println("Parent constructor");
    }

    public Parent(String name) {
        this.name = name;
        System.out.println("Parent constructor with name: " +
name);
    }

    public void display() {
        System.out.println("Parent display");
    }
}
class Child extends Parent {
    String name = "Child";

    public Child() {
        // Use 1: Call parent constructor
        super("Custom Parent");  // Must be first line
        System.out.println("Child constructor");
    }

    public void display() {
        // Use 2: Call parent method
        super.display();
        System.out.println("Child display");
    }

    public void showNames() {
        // Use 3: Access parent variable
        System.out.println("Child name: " + this.name);
        System.out.println("Parent name: " + super.name);
    }
}
public class Main {
    public static void main(String[] args) {
        Child child = new Child();
        System.out.println();

        child.display();
        System.out.println();

        child.showNames();
    }
}
```

Output:

```
Parent constructor with name: Custom Parent
Child constructor

Parent display
Child display

Child name: Child
Parent name: Custom Parent
```

Three Uses of super:

```
┌──────────────────────────────────────────┐
│ 1. super() - Call parent constructor     │
│    Must be first line in constructor     │
│                                          │
│ 2. super.method() - Call parent method   │
│    Used when method is overridden        │
│                                          │
│ 3. super.variable - Access parent var    │
│    Used when variable is hidden          │
└──────────────────────────────────────────┘
```

IS-A Relationship

Student: I've heard about "IS-A" relationship. What's that?

Expert: Inheritance represents an "IS-A" relationship. Let me explain:

```java
class Animal {
    void breathe() {
        System.out.println("Breathing...");
    }
}
class Dog extends Animal {
    void bark() {
        System.out.println("Woof!");
    }
}
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();

        // Dog IS-A Animal (inheritance relationship)
        // So we can do this:
        Animal animal = dog;  // Upcasting
        animal.breathe();     // Works fine
        // animal.bark();     // Compile error - Animal doesn't have bark()

        // We can check the relationship:
        System.out.println("dog instanceof Dog: " + (dog instanceof Dog));           // true
        System.out.println("dog instanceof Animal: " + (dog instanceof Animal));     // true
        System.out.println("animal instanceof Dog: " + (animal instanceof Dog));     // true
        System.out.println("animal instanceof Animal: " + (animal instanceof Animal)); // true
    }
}
```

IS-A Test:

```
If you can say "X IS-A Y", then X should extend Y

✅ Dog IS-A Animal          → Dog extends Animal
✅ Car IS-A Vehicle         → Car extends Vehicle
✅ Manager IS-A Employee    → Manager extends Employee

❌ Car IS-A Wheel           → Wrong! (Car HAS-A Wheel)
❌ House IS-A Room          → Wrong! (House HAS-A Room)
```

When to Use Inheritance

Student: How do I know when to use inheritance?

Expert: Use inheritance when:

```java
// ✅ GOOD use of inheritance - Clear IS-A relationship
class Shape {
    protected String color;

    public double calculateArea() {
        return 0;
    }
}
class Circle extends Shape {
    private double radius;

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}
class Rectangle extends Shape {
    private double length;
    private double width;

    @Override
    public double calculateArea() {
        return length * width;
    }
}
// Circle IS-A Shape ✅
// Rectangle IS-A Shape ✅
```

```java
// ❌ BAD use of inheritance - No IS-A relationship
class ArrayList {
    private Object[] elements;

    public void add(Object obj) {
        // Add logic
    }
}
// Stack IS-A ArrayList? No! This is wrong!
class Stack extends ArrayList {
    public void push(Object obj) {
        add(obj);
    }
}
// Better to use composition (HAS-A relationship)
```

Guidelines:

```
Use Inheritance When:

┌──────────────────────────────────────┐
│ ✅ Clear IS-A relationship exists    │
│ ✅ Child is a specialized parent     │
│ ✅ Need to reuse parent's code       │
│ ✅ Need polymorphic behavior         │
└──────────────────────────────────────┘

Avoid Inheritance When:

┌──────────────────────────────────────┐
│ ❌ Only want to reuse code (use      │
│    composition instead)              │
│ ❌ No clear IS-A relationship        │
│ ❌ Child is not a specialized parent │
│ ❌ Creates tight coupling            │
└──────────────────────────────────────┘
```

Interview Trap Alert! 🚨

Interviewer might ask: "Can you override a private method?"

Wrong Answer: "Yes, we can override private methods."

Correct Answer: “No, we cannot override private methods because private methods are not inherited. If a child class has a method with the same signature as a private method in the parent, it’s not overriding—it’s a completely new method.”

```java
class Parent {
    private void display() {
        System.out.println("Parent private display");
    }

    public void callDisplay() {
        display();  // Calls Parent's private display
    }
}
class Child extends Parent {
    // This is NOT overriding - it's a new method
    private void display() {
        System.out.println("Child private display");
    }

    public void callChildDisplay() {
        display();  // Calls Child's own display
    }
}
public class Main {
    public static void main(String[] args) {
        Child child = new Child();
        child.callDisplay();       // Parent private display
        child.callChildDisplay();  // Child private display
    }
}
```

Another Common Question: “Can you override static methods?”

Answer: “No, static methods cannot be overridden. They can be hidden (method hiding), but it’s not polymorphic.”

```java
class Parent {
    public static void staticMethod() {
        System.out.println("Parent static method");
    }
}
class Child extends Parent {
    // This is method HIDING, not overriding
    public static void staticMethod() {
        System.out.println("Child static method");
    }
}
public class Main {
    public static void main(String[] args) {
        Parent.staticMethod();  // Parent static method
        Child.staticMethod();   // Child static method

        Parent obj = new Child();
        obj.staticMethod();     // Parent static method (NOT polymorphic!)
    }
}
```

Chapter Recap

✅ Inheritance allows a class to acquire properties and methods of another class
✅ Use `extends` keyword for inheritance
✅ Child class inherits public and protected members
✅ Private members are NOT inherited
✅ Parent constructor is always called first
✅ Method overriding allows changing inherited behavior
✅ Use `super` to access parent class members
✅ Inheritance represents IS-A relationship
✅ Java doesn't support multiple inheritance with classes

Interview Questions

1. **Q:** What is inheritance?
   A: Inheritance is a mechanism where one class acquires the properties and behaviors of another class, promoting code reusability.

2. **Q:** What keyword is used for inheritance in Java?

A: The `extends` keyword is used for inheritance.

3. **Q:** What is method overriding?
   A: Method overriding is when a child class provides a specific implementation for a method that is already defined in its parent class.

4. **Q:** Can we override private methods?
   A: No, private methods are not inherited, so they cannot be overridden.

5. **Q:** Can we override static methods?
   A: No, static methods can be hidden but not overridden. They are not polymorphic.

6. **Q:** What is the use of the `super` keyword?
   A: `super` is used to call parent class constructors, methods, and access parent class variables.

7. **Q:** Why doesn't Java support multiple inheritance?
   A: To avoid the diamond problem where ambiguity arises when a class inherits from multiple classes with the same method.

8. **Q:** What is the difference between IS-A and HAS-A relationship?
   A: IS-A represents inheritance (Dog IS-A Animal), while HAS-A represents composition (Car HAS-A Engine).

Chapter 6: Polymorphism

Student–Expert Dialogue

Student: Polymorphism sounds complicated. What does it mean?

Expert: "Polymorphism" comes from Greek: "poly" = many, "morph" = forms. In programming, it means "one interface, many implementations." Think of it like a smartphone: - Same interface (touchscreen) - Many actions (call, text, browse, play games)

Student: Can you show me a code example?

Expert: Absolutely! Let's start with a simple example:

```java
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}
class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Woof! Woof!");
    }
}
class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Meow! Meow!");
    }
}
class Cow extends Animal {
    @Override
    void makeSound() {
        System.out.println("Moo! Moo!");
    }
}
public class Main {
    public static void main(String[] args) {
        // Same reference type (Animal), different objects
        Animal animal1 = new Dog();
        Animal animal2 = new Cat();
        Animal animal3 = new Cow();

        // Same method call, different behavior - POLYMORPHISM!
        animal1.makeSound();  // Woof! Woof!
        animal2.makeSound();  // Meow! Meow!
        animal3.makeSound();  // Moo! Moo!
    }
}
```

Output:

```
Woof! Woof!
Meow! Meow!
Moo! Moo!
```

Student: So the same method call produces different results?

Expert: Exactly! That's the power of polymorphism. Let me show you why this is useful:

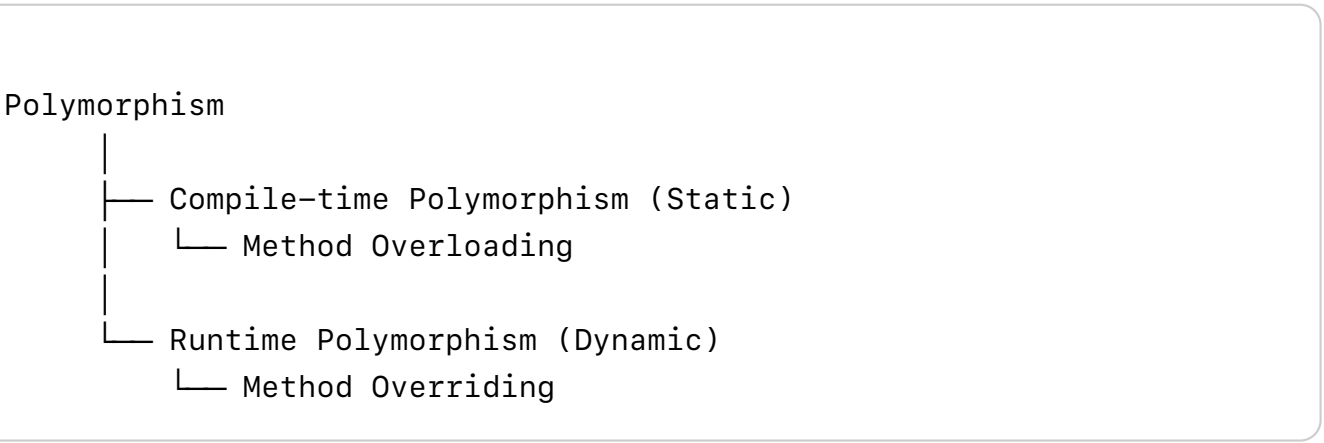
```java
class Animal {
    String name;

    Animal(String name) {
        this.name = name;
    }

    void makeSound() {
        System.out.println(name + " makes a sound");
    }

    void eat() {
        System.out.println(name + " is eating");
    }
}
class Dog extends Animal {
    Dog(String name) {
        super(name);
    }

    @Override
    void makeSound() {
        System.out.println(name + " says: Woof!");
    }
}
class Cat extends Animal {
    Cat(String name) {
        super(name);
    }

    @Override
    void makeSound() {
        System.out.println(name + " says: Meow!");
    }
}
public class Main {
    // One method works with all animal types!
    public static void makeAnimalSpeak(Animal animal) {
        animal.makeSound();  // Polymorphic call
    }

    public static void feedAnimal(Animal animal) {
        animal.eat();
```

```
    }

    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
        Cat cat = new Cat("Whiskers");

        // Same method, different animals
        makeAnimalSpeak(dog);  // Buddy says: Woof!
        makeAnimalSpeak(cat);  // Whiskers says: Meow!

        feedAnimal(dog);  // Buddy is eating
        feedAnimal(cat);  // Whiskers is eating
    }
}
```

Types of Polymorphism

Student: Are there different types of polymorphism?

Expert: Yes! There are two main types:



1. Compile-time Polymorphism (Method Overloading)

```java
class Calculator {
    // Same method name, different parameters

    // Method 1: Two integers
    int add(int a, int b) {
        return a + b;
    }

    // Method 2: Three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method 3: Two doubles
    double add(double a, double b) {
        return a + b;
    }

    // Method 4: String concatenation
    String add(String a, String b) {
        return a + b;
    }
}
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();

        System.out.println(calc.add(5, 10));           // 15 (Method 1)
        System.out.println(calc.add(5, 10, 15));       // 30 (Method 2)
        System.out.println(calc.add(5.5, 10.5));       // 16.0 (Method 3)
        System.out.println(calc.add("Hello", "World")); // HelloWorld (Method 4)
    }
}
```

Student: How does Java know which method to call?

Expert: The compiler decides at compile-time based on: 1. Number of parameters 2. Type of parameters 3. Order of parameters

```java
class Demo {
    void display(int a) {
        System.out.println("Integer: " + a);
    }

    void display(String a) {
        System.out.println("String: " + a);
    }

    void display(int a, String b) {
        System.out.println("Int and String: " + a + ", " + b);
    }

    void display(String a, int b) {
        System.out.println("String and Int: " + a + ", " + b);
    }
}
public class Main {
    public static void main(String[] args) {
        Demo demo = new Demo();

        demo.display(10);               // Calls display(int)
        demo.display("Hello");          // Calls display(String)
        demo.display(10, "World");      // Calls display(int, String)
        demo.display("Hello", 20);      // Calls display(String, int)
    }
}
```

2. Runtime Polymorphism (Method Overriding)

```java
class Shape {
    void draw() {
        System.out.println("Drawing a shape");
    }

    double calculateArea() {
        return 0;
    }
}
class Circle extends Shape {
    double radius;

```

```java
    Circle(double radius) {
        this.radius = radius;
    }

    @Override
    void draw() {
        System.out.println("Drawing a circle");
    }

    @Override
    double calculateArea() {
        return Math.PI * radius * radius;
    }
}
class Rectangle extends Shape {
    double length, width;

    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    @Override
    void draw() {
        System.out.println("Drawing a rectangle");
    }

    @Override
    double calculateArea() {
        return length * width;
    }
}
public class Main {
    public static void main(String[] args) {
        // Reference type: Shape
        // Object type: Circle, Rectangle (decided at runtime)
        Shape shape1 = new Circle(5);
        Shape shape2 = new Rectangle(4, 6);

        // Method call resolved at RUNTIME
        shape1.draw();  // Drawing a circle
        System.out.println("Area: " + shape1.calculateArea());
// 78.53...

        shape2.draw();  // Drawing a rectangle
        System.out.println("Area: " + shape2.calculateArea());
// 24.0
    }
```

```
62 }
```

Student: What's the difference between compile-time and runtime polymorphism?

Expert: Let me show you:

```
Compile-time Polymorphism          Runtime Polymorphism
┌──────────────────────────┐       ┌──────────────────────────┐
│ Method Overloading       │       │ Method Overriding        │
│                          │       │                          │
│ Same class               │       │ Different classes        │
│ Same method name         │       │ Same method name         │
│ Different parameters     │       │ Same parameters          │
│ Resolved at compile-time │       │ Resolved at runtime      │
│ Static binding           │       │ Dynamic binding          │
│ Faster                   │       │ Slower (but flexible)    │
└──────────────────────────┘       └──────────────────────────┘
```

Upcasting and Downcasting

Student: I've seen code like `Animal animal = new Dog()`. What's happening there?

Expert: That's called **upcasting**. Let me explain both upcasting and downcasting:

```java
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}
class Dog extends Animal {
    @Override
    void eat() {
        System.out.println("Dog is eating");
    }

    void bark() {
        System.out.println("Dog is barking");
    }
}
public class Main {
    public static void main(String[] args) {
```

```
        // UPCASTING - Implicit (automatic)
        // Child to Parent reference
        Animal animal = new Dog();  // Upcasting happens automatically

        animal.eat();   // Works - method is overridden
        // animal.bark();  // ❌ Compile error - Animal doesn't have bark()

        // DOWNCASTING - Explicit (manual)
        // Parent to Child reference
        Dog dog = (Dog) animal;  // Need explicit cast
        dog.eat();   // Works
        dog.bark();  // Works now!

        System.out.println();

        // ⚠️ DANGEROUS DOWNCASTING
        Animal animal2 = new Animal();  // Actually an Animal object
        // Dog dog2 = (Dog) animal2;  // ❌ ClassCastException at runtime!

        // SAFE DOWNCASTING - Check before casting
        if (animal2 instanceof Dog) {
            Dog dog2 = (Dog) animal2;
            dog2.bark();
        } else {
            System.out.println("animal2 is not a Dog!");
        }
    }
}
```

Output:

```
Dog is eating
Dog is eating
Dog is barking

animal2 is not a Dog!
```

Visual Representation:

```
Upcasting (Implicit):
    Dog ————————> Animal
    (Child)         (Parent)
    Automatic, Always Safe

Downcasting (Explicit):
    Animal ————————> Dog
    (Parent)         (Child)
    Manual, Can be Risky
```

Student: When should I use `instanceof`?

Expert: Always use `instanceof` before downcasting to avoid `ClassCastException`:

```java
class Animal {
    void makeSound() {
        System.out.println("Some sound");
    }
}
class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Woof!");
    }

    void fetch() {
        System.out.println("Fetching ball");
    }
}
class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Meow!");
    }

    void scratch() {
        System.out.println("Scratching");
    }
}
public class Main {
    public static void handleAnimal(Animal animal) {
        // Safe - available in Animal class
        animal.makeSound();
```

```java

        // Check type before calling specific methods
        if (animal instanceof Dog) {
            Dog dog = (Dog) animal;
            dog.fetch();
        } else if (animal instanceof Cat) {
            Cat cat = (Cat) animal;
            cat.scratch();
        }
    }

    public static void main(String[] args) {
        Animal dog = new Dog();
        Animal cat = new Cat();

        handleAnimal(dog);
        System.out.println();
        handleAnimal(cat);
    }
}
```

Output:

```
Woof!
Fetching ball

Meow!
Scratching
```

Real-World Example: Payment Processing System

Student: Can you show a complete real-world example?

Expert: Sure! Let's build a payment processing system:

```java
// Base class
abstract class Payment {
    protected String transactionId;
    protected double amount;
    protected String customerName;

    public Payment(String transactionId, double amount, String
```

```
customerName) {
        this.transactionId = transactionId;
        this.amount = amount;
        this.customerName = customerName;
    }

    // Template method - same for all payments
    public final void processPayment() {
        System.out.println("=== Processing Payment ===");
        System.out.println("Transaction ID: " + transactionId);
        System.out.println("Customer: " + customerName);
        System.out.println("Amount: $" + amount);

        if (validatePayment()) {
            if (executePayment()) {
                generateReceipt();
                sendConfirmation();
                System.out.println("Payment successful!");
            } else {
                System.out.println("Payment failed!");
            }
        } else {
            System.out.println("Payment validation failed!");
        }
        System.out.println();
    }

    // Abstract methods - different for each payment type
    protected abstract boolean validatePayment();
    protected abstract boolean executePayment();

    // Common methods
    private void generateReceipt() {
        System.out.println("Receipt generated");
    }

    private void sendConfirmation() {
        System.out.println("Confirmation email sent");
    }
}
// Credit Card Payment
class CreditCardPayment extends Payment {
    private String cardNumber;
    private String cvv;
    private String expiryDate;

    public CreditCardPayment(String transactionId, double amount,
String customerName,
```

```
                            String cardNumber, String cvv, String
expiryDate) {
        super(transactionId, amount, customerName);
        this.cardNumber = cardNumber;
        this.cvv = cvv;
        this.expiryDate = expiryDate;
    }

    @Override
    protected boolean validatePayment() {
        System.out.println("Validating credit card...");
        // Simplified validation
        return cardNumber.length() == 16 && cvv.length() == 3;
    }

    @Override
    protected boolean executePayment() {
        System.out.println("Charging credit card: ****-****-****-
" +
                           cardNumber.substring(12));
        return true;
    }
}
// PayPal Payment
class PayPalPayment extends Payment {
    private String email;
    private String password;

    public PayPalPayment(String transactionId, double amount,
String customerName,
                         String email, String password) {
        super(transactionId, amount, customerName);
        this.email = email;
        this.password = password;
    }

    @Override
    protected boolean validatePayment() {
        System.out.println("Validating PayPal account...");
        return email.contains("@") && password.length() >= 6;
    }

    @Override
    protected boolean executePayment() {
        System.out.println("Processing PayPal payment for: " +
email);
        return true;
    }
```

```java
}
// Bank Transfer Payment
class BankTransferPayment extends Payment {
    private String accountNumber;
    private String ifscCode;

    public BankTransferPayment(String transactionId, double amount, String customerName,
                               String accountNumber, String ifscCode) {
        super(transactionId, amount, customerName);
        this.accountNumber = accountNumber;
        this.ifscCode = ifscCode;
    }

    @Override
    protected boolean validatePayment() {
        System.out.println("Validating bank account...");
        return accountNumber.length() >= 10 && ifscCode.length() == 11;
    }

    @Override
    protected boolean executePayment() {
        System.out.println("Initiating bank transfer from account: " +
                           "****" + accountNumber.substring(accountNumber.length() - 4));
        return true;
    }
}
// Cryptocurrency Payment
class CryptoPayment extends Payment {
    private String walletAddress;
    private String cryptoType;

    public CryptoPayment(String transactionId, double amount, String customerName,
                         String walletAddress, String cryptoType) {
        super(transactionId, amount, customerName);
        this.walletAddress = walletAddress;
        this.cryptoType = cryptoType;
    }

    @Override
    protected boolean validatePayment() {
        System.out.println("Validating " + cryptoType + "
```

```
    wallet...");
143         return walletAddress.length() >= 26;
144     }
145
146     @Override
147     protected boolean executePayment() {
148         System.out.println("Processing " + cryptoType + "
    payment");
149         System.out.println("Wallet: " +
    walletAddress.substring(0, 10) + "...");
150         return true;
151     }
152 }
153
154 // Payment Processor
155 class PaymentProcessor {
156     // Polymorphic method - accepts any Payment type
157     public void process(Payment payment) {
158         payment.processPayment();
159     }
160
161     // Process multiple payments
162     public void processBatch(Payment[] payments) {
163         System.out.println("=== Batch Processing " +
    payments.length + " Payments ===\n");
164         double totalAmount = 0;
165
166         for (Payment payment : payments) {
167             payment.processPayment();
168             totalAmount += payment.amount;
169         }
170
171         System.out.println("Total amount processed: $" +
    totalAmount);
172     }
173 }
174
175 public class Main {
176     public static void main(String[] args) {
177         // Create different payment types
178         Payment creditCard = new CreditCardPayment(
179             "TXN001", 150.00, "John Doe",
180             "1234567890123456", "123", "12/25"
181         );
182
183         Payment paypal = new PayPalPayment(
184             "TXN002", 200.00, "Jane Smith",
185             "jane@example.com", "password123"
186         );
187
```

```
        Payment bankTransfer = new BankTransferPayment(
            "TXN003", 500.00, "Bob Johnson",
            "1234567890", "ABCD0123456"
        );

        Payment crypto = new CryptoPayment(
            "TXN004", 1000.00, "Alice Williams",
            "1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa", "Bitcoin"
        );

        // Process individual payments
        PaymentProcessor processor = new PaymentProcessor();

        processor.process(creditCard);
        processor.process(paypal);
        processor.process(bankTransfer);
        processor.process(crypto);

        // Batch processing
        Payment[] payments = {creditCard, paypal, bankTransfer,
    crypto};
        processor.processBatch(payments);
    }
}
```

Output:

```
=== Processing Payment ===
Transaction ID: TXN001
Customer: John Doe
Amount: $150.0
Validating credit card...
Charging credit card: ****-****-****-3456
Receipt generated
Confirmation email sent
Payment successful!

=== Processing Payment ===
Transaction ID: TXN002
Customer: Jane Smith
Amount: $200.0
Validating PayPal account...
Processing PayPal payment for: jane@example.com
Receipt generated
```

```
Confirmation email sent
Payment successful!

=== Processing Payment ===
Transaction ID: TXN003
Customer: Bob Johnson
Amount: $500.0
Validating bank account...
Initiating bank transfer from account: ****7890
Receipt generated
Confirmation email sent
Payment successful!

=== Processing Payment ===
Transaction ID: TXN004
Customer: Alice Williams
Amount: $1000.0
Validating Bitcoin wallet...
Processing Bitcoin payment
Wallet: 1A1zP1eP5Q...
Receipt generated
Confirmation email sent
Payment successful!

=== Batch Processing 4 Payments ===

[... repeated output ...]

Total amount processed: $1850.0
```

Dynamic Method Dispatch

Student: How does Java decide which method to call at runtime?

Expert: It's called **Dynamic Method Dispatch**. Let me explain:

```java
class Vehicle {
    void start() {
        System.out.println("Vehicle is starting");
    }

    void stop() {
```

```
        System.out.println("Vehicle is stopping");
    }
}
class Car extends Vehicle {
    @Override
    void start() {
        System.out.println("Car is starting with key");
    }

    // stop() is inherited, not overridden
}
class Bike extends Vehicle {
    @Override
    void start() {
        System.out.println("Bike is starting with kick");
    }

    @Override
    void stop() {
        System.out.println("Bike is stopping with hand brake");
    }
}
public class Main {
    public static void main(String[] args) {
        // Reference type: Vehicle
        // Object type: determined at runtime

        Vehicle v1 = new Car();
        Vehicle v2 = new Bike();
        Vehicle v3 = new Vehicle();

        System.out.println("=== v1 (Car object) ===");
        v1.start();  // Car's overridden method
        v1.stop();   // Vehicle's method (not overridden in Car)

        System.out.println("\n=== v2 (Bike object) ===");
        v2.start();  // Bike's overridden method
        v2.stop();   // Bike's overridden method

        System.out.println("\n=== v3 (Vehicle object) ===");
        v3.start();  // Vehicle's method
        v3.stop();   // Vehicle's method
    }
}
```

Output:

```
=== v1 (Car object) ===
Car is starting with key
Vehicle is stopping

=== v2 (Bike object) ===
Bike is starting with kick
Bike is stopping with hand brake

=== v3 (Vehicle object) ===
Vehicle is starting
Vehicle is stopping
```

How it works:

```
At Runtime:
┌─────────────────────────────────────┐
│ 1. Check actual object type         │
│    (not reference type)             │
│                                     │
│ 2. Look for method in object's      │
│    class                            │
│                                     │
│ 3. If not found, look in parent     │
│    class                            │
│                                     │
│ 4. Continue up the hierarchy        │
└─────────────────────────────────────┘
```

Polymorphism with Arrays

Student: Can I use polymorphism with arrays?

Expert: Absolutely! That's one of the most powerful uses:

```
1 abstract class Employee {
2     protected String name;
3     protected String id;
4
```

```
    public Employee(String name, String id) {
        this.name = name;
        this.id = id;
    }

    public abstract double calculateSalary();

    public void displayInfo() {
        System.out.println("Name: " + name + ", ID: " + id +
                           ", Salary: $" + calculateSalary());
    }
}
class Manager extends Employee {
    private double baseSalary;
    private double bonus;

    public Manager(String name, String id, double baseSalary,
double bonus) {
        super(name, id);
        this.baseSalary = baseSalary;
        this.bonus = bonus;
    }

    @Override
    public double calculateSalary() {
        return baseSalary + bonus;
    }
}
class Developer extends Employee {
    private double hourlyRate;
    private int hoursWorked;

    public Developer(String name, String id, double hourlyRate,
int hoursWorked) {
        super(name, id);
        this.hourlyRate = hourlyRate;
        this.hoursWorked = hoursWorked;
    }

    @Override
    public double calculateSalary() {
        return hourlyRate * hoursWorked;
    }
}
class Intern extends Employee {
    private double stipend;

    public Intern(String name, String id, double stipend) {
```

```java
        super(name, id);
        this.stipend = stipend;
    }

    @Override
    public double calculateSalary() {
        return stipend;
    }
}
public class Main {
    public static void main(String[] args) {
        // Polymorphic array - holds different employee types
        Employee[] employees = new Employee[5];

        employees[0] = new Manager("Alice", "M001", 8000, 2000);
        employees[1] = new Developer("Bob", "D001", 50, 160);
        employees[2] = new Developer("Charlie", "D002", 60, 150);
        employees[3] = new Intern("David", "I001", 1000);
        employees[4] = new Manager("Eve", "M002", 9000, 2500);

        // Process all employees polymorphically
        System.out.println("=== Employee Payroll ===");
        double totalPayroll = 0;

        for (Employee emp : employees) {
            emp.displayInfo();  // Polymorphic call
            totalPayroll += emp.calculateSalary();
        }

        System.out.println("\nTotal Payroll: $" + totalPayroll);

        // Count employees by type
        int managers = 0, developers = 0, interns = 0;

        for (Employee emp : employees) {
            if (emp instanceof Manager) {
                managers++;
            } else if (emp instanceof Developer) {
                developers++;
            } else if (emp instanceof Intern) {
                interns++;
            }
        }

        System.out.println("\n=== Employee Count ===");
        System.out.println("Managers: " + managers);
        System.out.println("Developers: " + developers);
        System.out.println("Interns: " + interns);
```

```
    }
}
```

Output:

```
=== Employee Payroll ===
Name: Alice, ID: M001, Salary: $10000.0
Name: Bob, ID: D001, Salary: $8000.0
Name: Charlie, ID: D002, Salary: $9000.0
Name: David, ID: I001, Salary: $1000.0
Name: Eve, ID: M002, Salary: $11500.0

Total Payroll: $39500.0

=== Employee Count ===
Managers: 2
Developers: 2
Interns: 1
```

Benefits of Polymorphism

Student: Why is polymorphism so important?

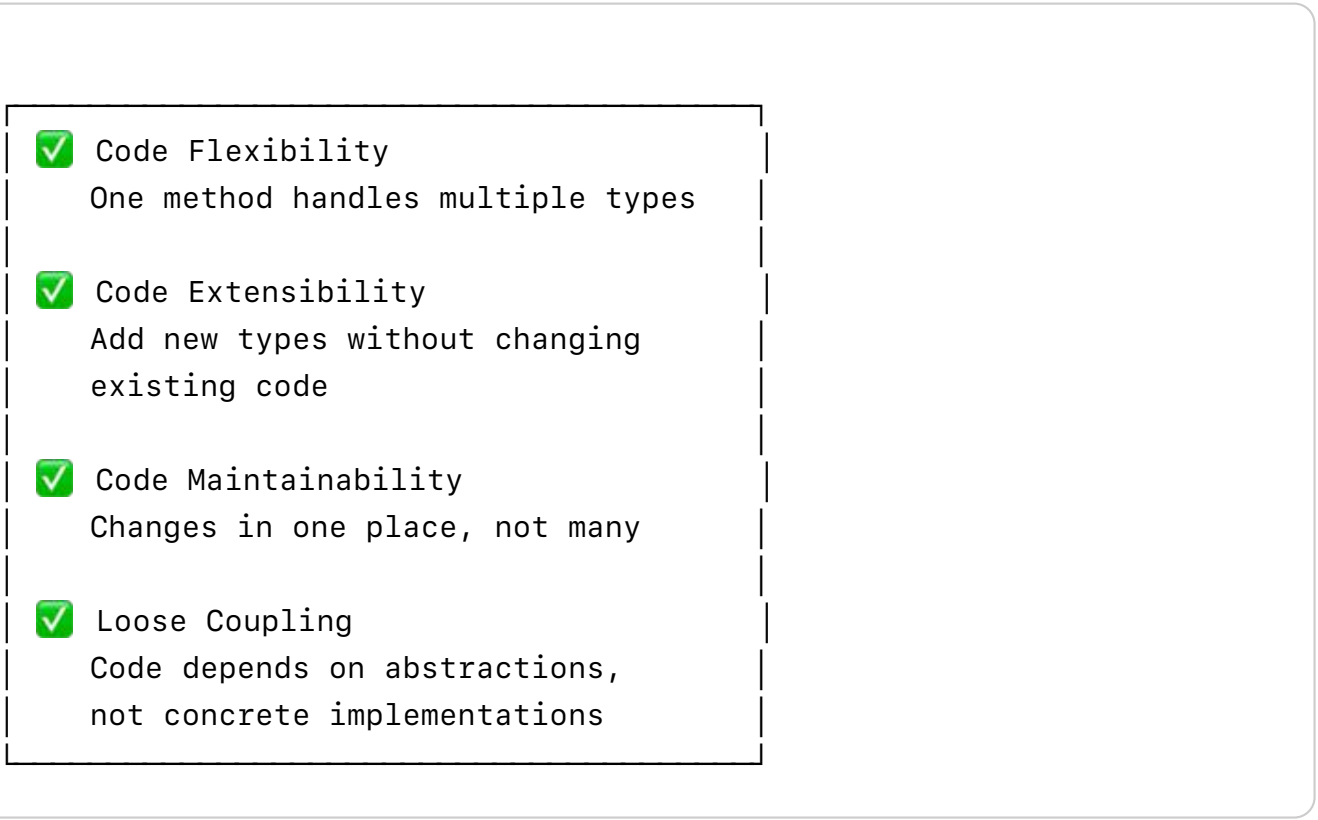
Expert: Let me show you with a before-and-after example:

```java
// WITHOUT Polymorphism - Rigid and Hard to Maintain
class ShapeProcessor_Bad {
    public void drawCircle(Circle circle) {
        System.out.println("Drawing circle");
    }

    public void drawRectangle(Rectangle rectangle) {
        System.out.println("Drawing rectangle");
    }

    public void drawTriangle(Triangle triangle) {
        System.out.println("Drawing triangle");
    }

    // Need to add new method for every new shape!
    // public void drawPentagon(Pentagon pentagon) { ... }
```

```java
}
// WITH Polymorphism - Flexible and Easy to Extend
abstract class Shape {
    abstract void draw();
}
class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing circle");
    }
}
class Rectangle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing rectangle");
    }
}
class Triangle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing triangle");
    }
}
// Can add new shapes without changing this class!
class Pentagon extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing pentagon");
    }
}
class ShapeProcessor_Good {
    // One method handles ALL shapes!
    public void drawShape(Shape shape) {
        shape.draw();  // Polymorphic call
    }

    // Can process multiple shapes
    public void drawAll(Shape[] shapes) {
        for (Shape shape : shapes) {
            shape.draw();
        }
    }
}
public class Main {
    public static void main(String[] args) {
        ShapeProcessor_Good processor = new
ShapeProcessor_Good();

```

```
        Shape[] shapes = {
            new Circle(),
            new Rectangle(),
            new Triangle(),
            new Pentagon()  // New shape - no changes needed!
        };

        processor.drawAll(shapes);
    }
}
```

Benefits:



Interview Trap Alert! 🚨

Interviewer might ask: "Can you achieve polymorphism without inheritance?"

Wrong Answer: "No, polymorphism requires inheritance."

Correct Answer: "Yes, through interfaces! Java supports polymorphism through both inheritance and interfaces."

```java
// Polymorphism with interfaces
interface Playable {
    void play();
}
class Guitar implements Playable {
    @Override
    public void play() {
        System.out.println("Playing guitar: Strum strum");
    }
}
class Piano implements Playable {
    @Override
    public void play() {
        System.out.println("Playing piano: Ding ding");
    }
}
class Drums implements Playable {
    @Override
    public void play() {
        System.out.println("Playing drums: Boom boom");
    }
}
public class Main {
    public static void playInstrument(Playable instrument) {
        instrument.play();  // Polymorphism!
    }

    public static void main(String[] args) {
        playInstrument(new Guitar());
        playInstrument(new Piano());
        playInstrument(new Drums());
    }
}
```

Another Common Question: “What’s the difference between overloading and overriding?”

	Overloading	Overriding
Class	Same class	Different class
Inheritance	Not required	Required
Method name	Same	Same
Parameters	Different	Same
Return type	Can be different	Same/covariant
Binding	Compile-time	Runtime
Polymorphism	Compile-time	Runtime

Chapter Recap

✅ Polymorphism = One interface, many implementations
✅ Two types: Compile-time (overloading) and Runtime (overriding)
✅ Enables writing flexible, extensible code
✅ Upcasting is automatic, downcasting requires explicit cast
✅ Use `instanceof` before downcasting
✅ Dynamic method dispatch resolves method calls at runtime
✅ Works with both inheritance and interfaces

Interview Questions

1. **Q:** What is polymorphism?
   A: Polymorphism means “many forms.” It allows one interface to be used for different data types or objects, enabling the same method call to behave differently based on the object.

2. **Q:** What are the types of polymorphism in Java?
   A: Compile-time polymorphism (method overloading) and runtime polymorphism (method overriding).

3. **Q:** What is the difference between overloading and overriding?
   A: Overloading is having multiple methods with the same name but different parameters in the same class. Overriding is redefining a parent class method in a child class with the same signature.

4. **Q:** What is upcasting and downcasting?

A: Upcasting is converting a child class reference to a parent class reference (automatic). Downcasting is converting a parent class reference to a child class reference (requires explicit casting).

5. **Q:** Why use polymorphism?
   A: For code flexibility, reusability, maintainability, and to write generic code that works with multiple types.

6. **Q:** Can we achieve polymorphism without inheritance?
   A: Yes, through interfaces.

7. **Q:** What is dynamic method dispatch?
   A: The mechanism by which a call to an overridden method is resolved at runtime based on the actual object type, not the reference type.

Chapter 7: Constructors

Student–Expert Dialogue

Student: We've used constructors in previous examples, but I'm not clear on what they really are.

Expert: A constructor is a special method that's automatically called when you create an object. Think of it as the "initialization method" for your object.

```java
class Student {
    String name;
    int rollNumber;

    // Constructor - same name as class, no return type
    Student() {
        System.out.println("Constructor called!");
        name = "Unknown";
        rollNumber = 0;
    }
}
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();  // Constructor is called here
        System.out.println("Name: " + s1.name);
        System.out.println("Roll Number: " + s1.rollNumber);
    }
}
```

Output:

```
Constructor called!
Name: Unknown
Roll Number: 0
```

Student: What makes constructors special?

Expert: Let me show you the key characteristics:

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

    // ✅ Constructor characteristics:
    // 1. Same name as class
    // 2. No return type (not even void)
    // 3. Called automatically when object is created
    Person() {
        System.out.println("Person constructor called");
    }

    // ❌ This is NOT a constructor (has return type)
    void Person() {
        System.out.println("This is a regular method, not a constructor!");
    }
}
public class Main {
    public static void main(String[] args) {
        Person p = new Person();  // Constructor called
        p.Person();  // Regular method called
    }
}
```

Types of Constructors

Student: Are there different types of constructors?

Expert: Yes! There are three main types:

1. Default Constructor (No-Argument)

```java
class Book {
    String title;
    String author;
    double price;

    // Default constructor - no parameters
    Book() {
        System.out.println("Default constructor called");
        title = "Unknown";
        author = "Unknown";
        price = 0.0;
    }

    void displayInfo() {
        System.out.println("Title: " + title);
        System.out.println("Author: " + author);
        System.out.println("Price: $" + price);
    }
}
public class Main {
    public static void main(String[] args) {
        Book book = new Book();
        book.displayInfo();
    }
}
```

Output:

```
Default constructor called
Title: Unknown
Author: Unknown
Price: $0.0
```

2. Parameterized Constructor

```java
class Book {
    String title;
    String author;
    double price;

    // Parameterized constructor
    Book(String t, String a, double p) {
        System.out.println("Parameterized constructor called");
        title = t;
        author = a;
        price = p;
    }

    void displayInfo() {
        System.out.println("Title: " + title);
        System.out.println("Author: " + author);
        System.out.println("Price: $" + price);
    }
}
public class Main {
    public static void main(String[] args) {
        Book book = new Book("Java Programming", "John Doe", 49.99);
        book.displayInfo();
    }
}
```

Output:

```
Parameterized constructor called
Title: Java Programming
Author: John Doe
Price: $49.99
```

3. Copy Constructor

```java
class Book {
    String title;
    String author;
    double price;
```

```java

    // Parameterized constructor
    Book(String title, String author, double price) {
        this.title = title;
        this.author = author;
        this.price = price;
    }

    // Copy constructor - creates a copy of another object
    Book(Book other) {
        System.out.println("Copy constructor called");
        this.title = other.title;
        this.author = other.author;
        this.price = other.price;
    }

    void displayInfo() {
        System.out.println("Title: " + title + ", Author: " +
author + ", Price: $" + price);
    }
}
public class Main {
    public static void main(String[] args) {
        Book original = new Book("Java Programming", "John Doe",
49.99);
        Book copy = new Book(original);  // Copy constructor

        System.out.println("Original: ");
        original.displayInfo();

        System.out.println("\nCopy: ");
        copy.displayInfo();

        // Modify copy
        copy.title = "Advanced Java";
        copy.price = 59.99;

        System.out.println("\nAfter modifying copy:");
        System.out.println("Original: ");
        original.displayInfo();

        System.out.println("\nCopy: ");
        copy.displayInfo();
    }
}
```

Output:

```
Copy constructor called
Original:
Title: Java Programming, Author: John Doe, Price: $49.99

Copy:
Title: Java Programming, Author: John Doe, Price: $49.99

After modifying copy:
Original:
Title: Java Programming, Author: John Doe, Price: $49.99

Copy:
Title: Advanced Java, Author: John Doe, Price: $59.99
```

Constructor Overloading

Student: Can we have multiple constructors in one class?

Expert: Absolutely! That's called constructor overloading:

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

    // Constructor 1: No parameters (default)
    Rectangle() {
        length = 1.0;
        width = 1.0;
        System.out.println("Default constructor: 1x1 rectangle created");
    }

    // Constructor 2: One parameter (square)
    Rectangle(double side) {
        length = side;
        width = side;
        System.out.println("Square constructor: " + side + "x" + side + " square created");
    }

    // Constructor 3: Two parameters (rectangle)
    Rectangle(double length, double width) {
```

```
        this.length = length;
        this.width = width;
        System.out.println("Rectangle constructor: " + length +
"x" + width + " rectangle created");
    }

    // Constructor 4: Copy constructor
    Rectangle(Rectangle other) {
        this.length = other.length;
        this.width = other.width;
        System.out.println("Copy constructor: copied rectangle");
    }

    double calculateArea() {
        return length * width;
    }

    void displayInfo() {
        System.out.println("Dimensions: " + length + " x " +
width);
        System.out.println("Area: " + calculateArea());
    }
}
public class Main {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();
        r1.displayInfo();
        System.out.println();

        Rectangle r2 = new Rectangle(5.0);
        r2.displayInfo();
        System.out.println();

        Rectangle r3 = new Rectangle(4.0, 6.0);
        r3.displayInfo();
        System.out.println();

        Rectangle r4 = new Rectangle(r3);
        r4.displayInfo();
    }
}
```

Output:

```
Default constructor: 1x1 rectangle created
Dimensions: 1.0 x 1.0
Area: 1.0

Square constructor: 5.0x5.0 square created
Dimensions: 5.0 x 5.0
Area: 25.0

Rectangle constructor: 4.0x6.0 rectangle created
Dimensions: 4.0 x 6.0
Area: 24.0

Copy constructor: copied rectangle
Dimensions: 4.0 x 6.0
Area: 24.0
```

Constructor Chaining with this()

Student: Can one constructor call another constructor?

Expert: Yes! That's called constructor chaining using `this()`:

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

    // Constructor 1: All parameters
    Employee(String name, int id, String department, double salary) {
        this.name = name;
        this.id = id;
        this.department = department;
        this.salary = salary;
        System.out.println("4-parameter constructor called");
    }

    // Constructor 2: Calls Constructor 1 with default department
    Employee(String name, int id, double salary) {
        this(name, id, "General", salary);  // Calls Constructor 1
```

```
        System.out.println("3-parameter constructor called");
    }

    // Constructor 3: Calls Constructor 2 with default salary
    Employee(String name, int id) {
        this(name, id, 30000.0);  // Calls Constructor 2
        System.out.println("2-parameter constructor called");
    }

    // Constructor 4: Calls Constructor 3 with default id
    Employee(String name) {
        this(name, 0);  // Calls Constructor 3
        System.out.println("1-parameter constructor called");
    }

    void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("ID: " + id);
        System.out.println("Department: " + department);
        System.out.println("Salary: $" + salary);
    }
}
public class Main {
    public static void main(String[] args) {
        System.out.println("=== Creating Employee 1 ===");
        Employee emp1 = new Employee("Alice", 101, "IT", 50000);
        emp1.displayInfo();

        System.out.println("\n=== Creating Employee 2 ===");
        Employee emp2 = new Employee("Bob", 102, 45000);
        emp2.displayInfo();

        System.out.println("\n=== Creating Employee 3 ===");
        Employee emp3 = new Employee("Charlie", 103);
        emp3.displayInfo();

        System.out.println("\n=== Creating Employee 4 ===");
        Employee emp4 = new Employee("David");
        emp4.displayInfo();
    }
}
```

Output:

```
=== Creating Employee 1 ===
4-parameter constructor called
Name: Alice
ID: 101
Department: IT
Salary: $50000.0

=== Creating Employee 2 ===
4-parameter constructor called
3-parameter constructor called
Name: Bob
ID: 102
Department: General
Salary: $45000.0

=== Creating Employee 3 ===
4-parameter constructor called
3-parameter constructor called
2-parameter constructor called
Name: Charlie
ID: 103
Department: General
Salary: $30000.0

=== Creating Employee 4 ===
4-parameter constructor called
3-parameter constructor called
2-parameter constructor called
1-parameter constructor called
Name: David
ID: 0
Department: General
Salary: $30000.0
```

Important Rules for this():

```
┌──────────────────────────────────────────┐
│ ✅ this() must be the FIRST statement    │
│    in the constructor                    │
│                                          │
│ ❌ Cannot have both this() and super()   │
│    in the same constructor               │
│                                          │
│ ❌ Cannot create circular chain          │
│    (A calls B, B calls A)                │
└──────────────────────────────────────────┘
```

Student: Can you show what happens with a circular chain?

Expert: Sure, but it won't compile:

```java
class BadExample {
    // ❌ This will cause a compile error
    /*
    BadExample() {
        this(10);  // Calls second constructor
    }

    BadExample(int x) {
        this();  // Calls first constructor - CIRCULAR!
    }
    // Error: recursive constructor invocation
    */
}
```

Default Constructor Behavior

Student: What if I don't write any constructor?

Expert: Java automatically provides a default constructor, but only if you don't write any constructor yourself:

```java
// Example 1: No constructor written
class Student1 {
    String name;
    int age;

    // Java automatically adds:
    // Student1() { }
}
// Example 2: Constructor written
class Student2 {
    String name;
    int age;

    Student2(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Java does NOT add default constructor
}
public class Main {
    public static void main(String[] args) {
        // ✅ Works - Java provided default constructor
        Student1 s1 = new Student1();

        // ✅ Works - using our constructor
        Student2 s2 = new Student2("Alice", 20);

        // ❌ Compile error - no default constructor in Student2
        // Student2 s3 = new Student2();
    }
}
```

Student: So if I write any constructor, I lose the default one?

Expert: Exactly! Let me show you a common pitfall:

```java
class BankAccount {
    String accountNumber;
    double balance;

    // We added a parameterized constructor
    BankAccount(String accountNumber, double balance) {
        this.accountNumber = accountNumber;
        this.balance = balance;
    }
}
class SavingsAccount extends BankAccount {
    double interestRate;

    SavingsAccount(String accountNumber, double balance, double interestRate) {
        // ❌ This will cause an error!
        // Java tries to call super() automatically
        // But BankAccount doesn't have a no-arg constructor anymore!

        // ✅ Fix: explicitly call parent's parameterized constructor
        super(accountNumber, balance);
        this.interestRate = interestRate;
    }
}
```

Private Constructors

Student: Can constructors be private?

Expert: Yes! Private constructors are used in specific design patterns:

```java
// Use Case 1: Singleton Pattern
class DatabaseConnection {
    private static DatabaseConnection instance;
    private String connectionString;

    // Private constructor - cannot create objects from outside
    private DatabaseConnection() {
        connectionString = "jdbc:mysql://localhost:3306/mydb";
        System.out.println("Database connection created");
    }
```

```java

    // Public method to get the single instance
    public static DatabaseConnection getInstance() {
        if (instance == null) {
            instance = new DatabaseConnection();
        }
        return instance;
    }

    public void connect() {
        System.out.println("Connecting to: " + connectionString);
    }
}
// Use Case 2: Utility Class (only static methods)
class MathUtils {
    // Private constructor - prevents instantiation
    private MathUtils() {
        throw new AssertionError("Cannot instantiate utility class");
    }

    public static int add(int a, int b) {
        return a + b;
    }

    public static int multiply(int a, int b) {
        return a * b;
    }
}
public class Main {
    public static void main(String[] args) {
        // ❌ Cannot do this - constructor is private
        // DatabaseConnection db = new DatabaseConnection();

        // ✅ Get singleton instance
        DatabaseConnection db1 = DatabaseConnection.getInstance();
        DatabaseConnection db2 = DatabaseConnection.getInstance();

        System.out.println("db1 == db2: " + (db1 == db2));  // true - same instance

        db1.connect();

        // ❌ Cannot do this - constructor is private
        // MathUtils utils = new MathUtils();

```

```
57          // ✅ Use static methods directly
58          System.out.println("5 + 3 = " + MathUtils.add(5, 3));
59          System.out.println("5 * 3 = " + MathUtils.multiply(5,
    3));
60      }
61  }
```

Output:

```
Database connection created
db1 == db2: true
Connecting to: jdbc:mysql://localhost:3306/mydb
5 + 3 = 8
5 * 3 = 15
```

Constructor vs Method

Student: What's the difference between constructors and regular methods?

Expert: Let me show you a comparison:

```java
class Demo {
    String name;

    // CONSTRUCTOR
    Demo(String name) {
        this.name = name;
        System.out.println("Constructor called");
    }

    // REGULAR METHOD
    void setName(String name) {
        this.name = name;
        System.out.println("Method called");
    }
}
public class Main {
    public static void main(String[] args) {
        Demo obj = new Demo("Initial");  // Constructor called automatically
        obj.setName("Updated");          // Method called explicitly
    }
}
```

Comparison Table:

Feature	Constructor	Method
Name	Same as class	Any name
Return type	None	Must have
Called	Automatically	Explicitly
When called	Object creation	Anytime
Purpose	Initialize object	Perform operations
Inheritance	Not inherited	Inherited
Can be static	No	Yes
Can be final	No	Yes
Can be abstract	No	Yes

Real-World Example: Building a Car

Student: Can you show a complete real-world example?

Expert: Sure! Let's build a car configuration system:

```
class Car {
    // Required fields
    private String brand;
    private String model;
    private int year;

    // Optional fields
    private String color;
    private String engineType;
    private boolean hasSunroof;
    private boolean hasLeatherSeats;
    private boolean hasNavigationSystem;

    // Constructor 1: Minimum required fields
    public Car(String brand, String model, int year) {
        this.brand = brand;
        this.model = model;
        this.year = year;
        // Optional fields get default values
        this.color = "White";
        this.engineType = "Petrol";
        this.hasSunroof = false;
        this.hasLeatherSeats = false;
        this.hasNavigationSystem = false;
        System.out.println("Basic car created");
    }

    // Constructor 2: With color
    public Car(String brand, String model, int year, String color) {
        this(brand, model, year);  // Call Constructor 1
        this.color = color;
        System.out.println("Car with custom color created");
    }

    // Constructor 3: With color and engine type
    public Car(String brand, String model, int year, String color, String engineType) {
        this(brand, model, year, color);  // Call Constructor 2
        this.engineType = engineType;
```

```
        System.out.println("Car with custom engine created");
    }

    // Constructor 4: Full customization
    public Car(String brand, String model, int year, String color, String engineType,
               boolean hasSunroof, boolean hasLeatherSeats, boolean hasNavigationSystem) {
        this(brand, model, year, color, engineType);  // Call Constructor 3
        this.hasSunroof = hasSunroof;
        this.hasLeatherSeats = hasLeatherSeats;
        this.hasNavigationSystem = hasNavigationSystem;
        System.out.println("Fully customized car created");
    }

    // Copy constructor
    public Car(Car other) {
        this.brand = other.brand;
        this.model = other.model;
        this.year = other.year;
        this.color = other.color;
        this.engineType = other.engineType;
        this.hasSunroof = other.hasSunroof;
        this.hasLeatherSeats = other.hasLeatherSeats;
        this.hasNavigationSystem = other.hasNavigationSystem;
        System.out.println("Car copied");
    }

    public void displaySpecs() {
        System.out.println("=== Car Specifications ===");
        System.out.println("Brand: " + brand);
        System.out.println("Model: " + model);
        System.out.println("Year: " + year);
        System.out.println("Color: " + color);
        System.out.println("Engine: " + engineType);
        System.out.println("Sunroof: " + (hasSunroof ? "Yes" : "No"));
        System.out.println("Leather Seats: " + (hasLeatherSeats ? "Yes" : "No"));
        System.out.println("Navigation: " + (hasNavigationSystem ? "Yes" : "No"));
        System.out.println("Price: $" + calculatePrice());
        System.out.println();
    }

    private double calculatePrice() {
        double basePrice = 20000;
```

```
81
82          if (engineType.equals("Diesel")) basePrice += 2000;
83          if (engineType.equals("Electric")) basePrice += 5000;
84          if (hasSunroof) basePrice += 1500;
85          if (hasLeatherSeats) basePrice += 2000;
86          if (hasNavigationSystem) basePrice += 1000;
87
88          return basePrice;
89      }
90  }
91  public class Main {
92      public static void main(String[] args) {
93          // Basic car
94          System.out.println("=== Car 1 ===");
95          Car car1 = new Car("Toyota", "Camry", 2023);
96          car1.displaySpecs();
97
98          // Car with custom color
99          System.out.println("=== Car 2 ===");
100         Car car2 = new Car("Honda", "Accord", 2023, "Blue");
101         car2.displaySpecs();
102
103         // Car with custom color and engine
104         System.out.println("=== Car 3 ===");
105         Car car3 = new Car("BMW", "X5", 2023, "Black", "Diesel");
106         car3.displaySpecs();
107
108         // Fully customized car
109         System.out.println("=== Car 4 ===");
110         Car car4 = new Car("Tesla", "Model S", 2023, "Red",
    "Electric",
111                            true, true, true);
112         car4.displaySpecs();
113
114         // Copy car
115         System.out.println("=== Car 5 (Copy of Car 4) ===");
116         Car car5 = new Car(car4);
117         car5.displaySpecs();
118     }
119 }
```

Output:

```
=== Car 1 ===
Basic car created
```

```
=== Car Specifications ===
Brand: Toyota
Model: Camry
Year: 2023
Color: White
Engine: Petrol
Sunroof: No
Leather Seats: No
Navigation: No
Price: $20000.0

=== Car 2 ===
Basic car created
Car with custom color created
=== Car Specifications ===
Brand: Honda
Model: Accord
Year: 2023
Color: Blue
Engine: Petrol
Sunroof: No
Leather Seats: No
Navigation: No
Price: $20000.0

=== Car 3 ===
Basic car created
Car with custom color created
Car with custom engine created
=== Car Specifications ===
Brand: BMW
Model: X5
Year: 2023
Color: Black
Engine: Diesel
Sunroof: No
Leather Seats: No
Navigation: No
Price: $22000.0

=== Car 4 ===
Basic car created
Car with custom color created
Car with custom engine created
```

```
Fully customized car created
=== Car Specifications ===
Brand: Tesla
Model: Model S
Year: 2023
Color: Red
Engine: Electric
Sunroof: Yes
Leather Seats: Yes
Navigation: Yes
Price: $29500.0

=== Car 5 (Copy of Car 4) ===
Car copied
=== Car Specifications ===
Brand: Tesla
Model: Model S
Year: 2023
Color: Red
Engine: Electric
Sunroof: Yes
Leather Seats: Yes
Navigation: Yes
Price: $29500.0
```

Interview Trap Alert! 🚨

Interviewer might ask: “Can a constructor be inherited?”

Wrong Answer: “Yes, constructors are inherited like other methods.”

Correct Answer: “No, constructors are NOT inherited. However, a child class constructor must call a parent class constructor (explicitly or implicitly) to initialize the inherited fields.”

```java
class Parent {
    int x;

    Parent(int x) {
        this.x = x;
        System.out.println("Parent constructor");
    }
}
class Child extends Parent {
    int y;

    // Child does NOT inherit Parent's constructor
    // Must explicitly call it
    Child(int x, int y) {
        super(x);  // Must call parent constructor
        this.y = y;
        System.out.println("Child constructor");
    }
}
public class Main {
    public static void main(String[] args) {
        Child c = new Child(10, 20);
    }
}
```

Another Common Question: “Can a constructor be final, static, or abstract?”

Answer:

```
┌──────────────────────────────────────┐
│ ❌ final constructor – NO            │
│    Constructors are not inherited,   │
│    so no need to prevent overriding  │
│                                      │
│ ❌ static constructor – NO           │
│    Constructors are called on objects,│
│    not on the class                  │
│                                      │
│ ❌ abstract constructor – NO         │
│    Constructors must have implementation │
└──────────────────────────────────────┘
```

Chapter Recap

✅ Constructors initialize objects when created
✅ Same name as class, no return type
✅ Three types: default, parameterized, copy
✅ Constructor overloading allows multiple constructors
✅ Use `this()` for constructor chaining
✅ Java provides default constructor only if you don't write any
✅ Private constructors used for Singleton and utility classes
✅ Constructors are NOT inherited

Interview Questions

1. **Q:** What is a constructor?
   A: A special method with the same name as the class, no return type, automatically called when an object is created to initialize it.

2. **Q:** What are the types of constructors?
   A: Default (no-argument), parameterized, and copy constructors.

3. **Q:** What is constructor overloading?
   A: Having multiple constructors in a class with different parameter lists.

4. **Q:** What is constructor chaining?
   A: When one constructor calls another constructor using `this()` or `super()`.

5. **Q:** Can a constructor be private?
   A: Yes, used in Singleton pattern and utility classes to prevent instantiation.

6. **Q:** Are constructors inherited?
   A: No, but child class constructors must call parent class constructors.

7. **Q:** What happens if you don't define any constructor?
   A: Java automatically provides a default no-argument constructor.

8. **Q:** Can a constructor return a value?
   A: Constructors don't have a return type, not even void. They implicitly return the newly created object.

9. **Q:** What is the difference between constructor and method?
   A: Constructors initialize objects, have no return type, same name as class, called automatically. Methods perform operations, have return type, any name, called explicitly.

Chapter 8: The `this` Keyword

Student–Expert Dialogue

Student: I keep seeing the `this` keyword everywhere. What exactly is it?

Expert: `this` is a reference to the current object. Think of it as the object saying "me" or "myself." Let me show you:

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

    void setDetails(String name, int age) {
        // Without 'this', this is confusing:
        // name = name;  // Which name? Parameter or field?

        // With 'this', it's clear:
        this.name = name;  // this.name is the field, name is the parameter
        this.age = age;
    }

    void display() {
        System.out.println("Name: " + this.name);  // 'this' is optional here
        System.out.println("Age: " + this.age);
    }
}
public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.setDetails("Alice", 25);
        p.display();
    }
}
```

Output:

```
Name: Alice
Age: 25
```

Uses of `this` Keyword

Student: What are all the ways I can use `this`?

Expert: There are 6 main uses. Let me show you each one:

Use 1: Distinguish Between Instance Variables and Parameters

```java
class Student {
    String name;
    int rollNumber;

    // Without 'this' - CONFUSING
    void setDetails_Bad(String name, int rollNumber) {
        name = name;  // Assigns parameter to itself!
        rollNumber = rollNumber;  // Assigns parameter to itself!
        // Instance variables remain unchanged!
    }

    // With 'this' - CLEAR
    void setDetails_Good(String name, int rollNumber) {
        this.name = name;  // Assigns parameter to instance variable
        this.rollNumber = rollNumber;
    }

    void display() {
        System.out.println("Name: " + name + ", Roll: " + rollNumber);
    }
}
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.setDetails_Bad("Alice", 101);
        s1.display();  // Name: null, Roll: 0 (not set!)

        Student s2 = new Student();
        s2.setDetails_Good("Bob", 102);
        s2.display();  // Name: Bob, Roll: 102 (correctly set!)
    }
}
```

Use 2: Call Current Class Method

```java
class Calculator {
    int result = 0;

    void add(int a, int b) {
        result = a + b;
        this.display();  // Calling current class method
        // display(); would also work, but 'this' makes it explicit
    }

    void subtract(int a, int b) {
        result = a - b;
        this.display();
    }

    void display() {
        System.out.println("Result: " + result);
    }
}
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.add(10, 5);
        calc.subtract(10, 5);
    }
}
```

Use 3: Call Current Class Constructor (Constructor Chaining)

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

    // Constructor 1
    Employee() {
        this("Unknown", 0, "General", 0.0);  // Calls Constructor 4
        System.out.println("Default constructor");
    }

    // Constructor 2
    Employee(String name, int id) {
```

```
        this(name, id, "General", 30000.0);  // Calls Constructor 4
        System.out.println("2-parameter constructor");
    }

    // Constructor 3
    Employee(String name, int id, String department) {
        this(name, id, department, 30000.0);  // Calls Constructor 4
        System.out.println("3-parameter constructor");
    }

    // Constructor 4
    Employee(String name, int id, String department, double salary) {
        this.name = name;
        this.id = id;
        this.department = department;
        this.salary = salary;
        System.out.println("4-parameter constructor");
    }

    void display() {
        System.out.println(name + " | " + id + " | " + department + " | $" + salary);
    }
}
public class Main {
    public static void main(String[] args) {
        Employee e1 = new Employee();
        e1.display();

        System.out.println();

        Employee e2 = new Employee("Alice", 101);
        e2.display();
    }
}
```

Output:

```
4-parameter constructor
Default constructor
Unknown | 0 | General | $0.0

4-parameter constructor
2-parameter constructor
Alice | 101 | General | $30000.0
```

Use 4: Pass Current Object as Parameter

```java
class Student {
    String name;
    int marks;

    Student(String name, int marks) {
        this.name = name;
        this.marks = marks;
    }

    void compareWith(Student other) {
        if (this.marks > other.marks) {
            System.out.println(this.name + " scored more than " + other.name);
        } else if (this.marks < other.marks) {
            System.out.println(this.name + " scored less than " + other.name);
        } else {
            System.out.println(this.name + " and " + other.name + " scored equal marks");
        }
    }

    void printDetails() {
        // Passing 'this' as parameter to another class method
        Printer.print(this);
    }
}
class Printer {
    static void print(Student student) {
        System.out.println("Student: " + student.name + ", Marks: " + student.marks);
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Alice", 85);
        Student s2 = new Student("Bob", 90);

        s1.compareWith(s2);
        s2.compareWith(s1);

        System.out.println();

        s1.printDetails();
        s2.printDetails();
    }
}
```

Output:

```
Alice scored less than Bob
Bob scored more than Alice

Student: Alice, Marks: 85
Student: Bob, Marks: 90
```

Use 5: Return Current Object

```java
class Builder {
    private String name;
    private int age;
    private String city;

    // Method chaining using 'this'
    Builder setName(String name) {
        this.name = name;
        return this;  // Returns current object
    }

    Builder setAge(int age) {
        this.age = age;
        return this;  // Returns current object
    }

    Builder setCity(String city) {
        this.city = city;
        return this;  // Returns current object
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " + age + ", City: " + city);
    }
}
public class Main {
    public static void main(String[] args) {
        Builder builder = new Builder();

        // Method chaining - possible because each method returns 'this'
        builder.setName("Alice")
               .setAge(25)
               .setCity("New York")
               .display();

        // Same as:
        // builder.setName("Alice");
        // builder.setAge(25);
        // builder.setCity("New York");
        // builder.display();
    }
}
```

Output:

```
Name: Alice, Age: 25, City: New York
```

Use 6: Pass as Argument in Constructor Call

```java
class Address {
    String city;
    String country;

    Address(String city, String country) {
        this.city = city;
        this.country = country;
    }

    void display() {
        System.out.println("Address: " + city + ", " + country);
    }
}
class Person {
    String name;
    Address address;

    Person(String name, String city, String country) {
        this.name = name;
        // Passing 'this' is not needed here, but showing the concept
        this.address = new Address(city, country);
    }

    void display() {
        System.out.println("Person: " + name);
        address.display();
    }
}
// Better example showing 'this' in constructor
class Employee {
    String name;
    EmployeeDetails details;

    Employee(String name, int id) {
        this.name = name;
        // Passing current Employee object to EmployeeDetails
        this.details = new EmployeeDetails(this, id);
    }
```

```
41
42      void display() {
43          System.out.println("Employee: " + name);
44          details.display();
45      }
46  }
47  class EmployeeDetails {
48      Employee employee;  // Reference to Employee object
49      int id;
50
51      EmployeeDetails(Employee employee, int id) {
52          this.employee = employee;
53          this.id = id;
54      }
55
56      void display() {
57          System.out.println("ID: " + id + ", Name: " +
    employee.name);
58      }
59  }
60  public class Main {
61      public static void main(String[] args) {
62          Employee emp = new Employee("Alice", 101);
63          emp.display();
64      }
65  }
```

Output:

```
Employee: Alice
ID: 101, Name: Alice
```

Real-World Example: Banking System

Student: Can you show a complete real-world example using `this`?

Expert: Sure! Let's build a banking system:

```
1  class BankAccount {
2      private String accountNumber;
3      private String holderName;
4      private double balance;
```

```
    private String accountType;

    // Use 1: Distinguish parameters from instance variables
    public BankAccount(String accountNumber, String holderName,
double balance, String accountType) {
        this.accountNumber = accountNumber;
        this.holderName = holderName;
        this.balance = balance;
        this.accountType = accountType;
    }

    // Use 3: Constructor chaining
    public BankAccount(String accountNumber, String holderName) {
        this(accountNumber, holderName, 0.0, "Savings");  //
Calls main constructor
    }

    // Use 5: Return current object for method chaining
    public BankAccount deposit(double amount) {
        if (amount > 0) {
            this.balance += amount;
            System.out.println("Deposited: $" + amount);
            this.printBalance();  // Use 2: Call current class
method
        }
        return this;  // Return current object
    }

    public BankAccount withdraw(double amount) {
        if (amount > 0 && amount <= this.balance) {
            this.balance -= amount;
            System.out.println("Withdrawn: $" + amount);
            this.printBalance();
        } else {
            System.out.println("Insufficient balance or invalid
amount!");
        }
        return this;  // Return current object
    }

    // Use 4: Pass current object as parameter
    public void transferTo(BankAccount recipient, double amount)
{
        if (amount > 0 && amount <= this.balance) {
            this.balance -= amount;
            recipient.balance += amount;
            System.out.println("Transferred $" + amount + " from
" +
```

```
                            this.holderName + " to " +
recipient.holderName);
            this.printBalance();
            recipient.printBalance();
        } else {
            System.out.println("Transfer failed!");
        }
    }

    public void compareBalance(BankAccount other) {
        if (this.balance > other.balance) {
            System.out.println(this.holderName + " has more
balance than " + other.holderName);
        } else if (this.balance < other.balance) {
            System.out.println(this.holderName + " has less
balance than " + other.holderName);
        } else {
            System.out.println(this.holderName + " and " +
other.holderName + " have equal balance");
        }
    }

    private void printBalance() {
        System.out.println(this.holderName + "'s balance: $" +
this.balance);
    }

    public void displayInfo() {
        System.out.println("=== Account Information ===");
        System.out.println("Account Number: " +
this.accountNumber);
        System.out.println("Holder Name: " + this.holderName);
        System.out.println("Account Type: " + this.accountType);
        System.out.println("Balance: $" + this.balance);
        System.out.println();
    }
}
public class Main {
    public static void main(String[] args) {
        // Using constructor chaining
        BankAccount account1 = new BankAccount("ACC001",
"Alice");
        BankAccount account2 = new BankAccount("ACC002", "Bob",
5000, "Current");

        account1.displayInfo();
        account2.displayInfo();

```

```
88          // Method chaining using 'this'
89          System.out.println("=== Method Chaining Demo ===");
90          account1.deposit(1000)
91                  .deposit(500)
92                  .withdraw(200);
93
94          System.out.println();
95
96          // Transfer between accounts
97          System.out.println("=== Transfer Demo ===");
98          account2.transferTo(account1, 1000);
99
100         System.out.println();
101
102         // Compare balances
103         System.out.println("=== Comparison Demo ===");
104         account1.compareBalance(account2);
105     }
106 }
```

Output:

```
=== Account Information ===
Account Number: ACC001
Holder Name: Alice
Account Type: Savings
Balance: $0.0

=== Account Information ===
Account Number: ACC002
Holder Name: Bob
Account Type: Current
Balance: $5000.0

=== Method Chaining Demo ===
Deposited: $1000.0
Alice's balance: $1000.0
Deposited: $500.0
Alice's balance: $1500.0
Withdrawn: $200.0
Alice's balance: $1300.0

=== Transfer Demo ===
Transferred $1000.0 from Bob to Alice
Bob's balance: $4000.0
Alice's balance: $2300.0

=== Comparison Demo ===
Alice has less balance than Bob
```

Common Mistakes with `this`

Student: What are some common mistakes people make with `this`?

Expert: Let me show you:

Mistake 1: Using `this` in Static Context

```java
class Demo {
    static int staticVar = 10;
    int instanceVar = 20;

    static void staticMethod() {
        System.out.println("Static var: " + staticVar);  // ✅ OK

        // ❌ Cannot use 'this' in static context
        // System.out.println("Instance var: " + this.instanceVar);  // Compile error!

        // Why? 'this' refers to current object, but static methods
        // don't belong to any object — they belong to the class
    }

    void instanceMethod() {
        System.out.println("Static var: " + staticVar);      // ✅ OK
        System.out.println("Instance var: " + this.instanceVar); // ✅ OK
    }
}
public class Main {
    public static void main(String[] args) {
        Demo.staticMethod();  // Called on class

        Demo obj = new Demo();
        obj.instanceMethod();  // Called on object
    }
}
```

Mistake 2: Forgetting `this` When Needed

```java
class Person {
    String name;

    // ❌ Bug: Parameter shadows instance variable
    void setName(String name) {
        name = name;  // Assigns parameter to itself!
        // Instance variable 'name' remains unchanged
    }

    // ✅ Correct: Use 'this' to access instance variable
    void setNameCorrect(String name) {
        this.name = name;
    }

    void display() {
        System.out.println("Name: " + name);
    }
}
public class Main {
    public static void main(String[] args) {
        Person p1 = new Person();
        p1.setName("Alice");
        p1.display();  // Name: null (Bug!)

        Person p2 = new Person();
        p2.setNameCorrect("Bob");
        p2.display();  // Name: Bob (Correct!)
    }
}
```

Mistake 3: Circular Reference with `this()`

```java
class BadExample {
    // ❌ This causes infinite recursion
    /*
    BadExample() {
        this(10);  // Calls second constructor
    }

    BadExample(int x) {
        this();  // Calls first constructor - CIRCULAR!
    }
    // Compile error: recursive constructor invocation
    */
}
```

Mistake 4: Using `this()` Not as First Statement

```java
class WrongConstructor {
    int x;

    // ❌ this() must be first statement
    /*
    WrongConstructor(int x) {
        this.x = x;
        this();  // Compile error: call to this must be first statement
    }

    WrongConstructor() {
        System.out.println("Default constructor");
    }
    */

    // ✅ Correct way
    WrongConstructor(int x) {
        this();  // Must be first
        this.x = x;
    }

    WrongConstructor() {
        System.out.println("Default constructor");
    }
}
```

`this` vs `super`

Student: What's the difference between `this` and `super`?

Expert: Great question! Let me show you:

```java
class Parent {
    String name = "Parent";

    Parent() {
        System.out.println("Parent constructor");
    }

    Parent(String name) {
        this.name = name;
        System.out.println("Parent parameterized constructor");
    }

    void display() {
        System.out.println("Parent display: " + name);
    }
}
class Child extends Parent {
    String name = "Child";

    Child() {
        super();  // Calls Parent's no-arg constructor
        System.out.println("Child constructor");
    }

    Child(String parentName, String childName) {
        super(parentName);  // Calls Parent's parameterized constructor
        this.name = childName;  // 'this' refers to Child's name
        System.out.println("Child parameterized constructor");
    }

    void display() {
        System.out.println("Child display: " + name);
    }

    void showBoth() {
        System.out.println("Child name (this): " + this.name);
        System.out.println("Parent name (super): " + super.name);
    }

```

```
    void callBothDisplays() {
        this.display();   // Calls Child's display
        super.display();  // Calls Parent's display
    }
}
public class Main {
    public static void main(String[] args) {
        System.out.println("=== Creating Child with default constructor ===");
        Child c1 = new Child();
        c1.showBoth();

        System.out.println("\n=== Creating Child with parameterized constructor ===");
        Child c2 = new Child("CustomParent", "CustomChild");
        c2.showBoth();

        System.out.println("\n=== Calling both displays ===");
        c2.callBothDisplays();
    }
}
```

Output:

```
=== Creating Child with default constructor ===
Parent constructor
Child constructor
Child name (this): Child
Parent name (super): Parent

=== Creating Child with parameterized constructor ===
Parent parameterized constructor
Child parameterized constructor
Child name (this): CustomChild
Parent name (super): CustomParent

=== Calling both displays ===
Child display: CustomChild
Parent display: CustomParent
```

Comparison:

Feature	this	super
Refers to	Current object	Parent object
Access variables	Current class	Parent class
Call methods	Current class	Parent class
Call constructor	Current class	Parent class
Usage	this.variable	super.variable
	this.method()	super.method()
	this()	super()

Interview Trap Alert! 🚨

Interviewer might ask: "Can we use both `this()` and `super()` in the same constructor?"

Wrong Answer: "Yes, we can use both."

Correct Answer: "No, we cannot use both `this()` and `super()` in the same constructor because both must be the first statement, and a constructor can have only one first statement."

```java
class Parent {
    Parent() {
        System.out.println("Parent constructor");
    }
}
class Child extends Parent {
    // ❌ Cannot have both
    /*
    Child() {
        super();  // First statement
        this(10); // Also wants to be first statement - ERROR!
    }
    */

    // ✅ Choose one
    Child() {
        super();  // Calls parent constructor
        System.out.println("Child no-arg constructor");
    }

    Child(int x) {
        this();  // Calls Child's no-arg constructor (which calls super)
        System.out.println("Child parameterized constructor");
    }
}
```

Another Common Question: "Can we use `this` keyword to refer to static variables?"

Answer: "Yes, technically you can, but it's not recommended. Static variables belong to the class, not to instances, so they should be accessed using the class name."

```java
class Demo {
    static int staticVar = 10;
    int instanceVar = 20;

    void display() {
        // ✅ Works but not recommended
        System.out.println(this.staticVar);

        // ✅ Recommended way
        System.out.println(Demo.staticVar);

        // ✅ This is fine
        System.out.println(this.instanceVar);
    }
}
```

Chapter Recap

✅ `this` refers to the current object
✅ Six main uses: distinguish variables, call methods, constructor chaining, pass as parameter, return object, pass in constructor
✅ Cannot use `this` in static context
✅ `this()` must be first statement in constructor
✅ Cannot use both `this()` and `super()` in same constructor
✅ `this` refers to current class, `super` refers to parent class

Interview Questions

1. **Q:** What is the `this` keyword?
   A: `this` is a reference variable that refers to the current object.

2. **Q:** What are the uses of `this` keyword?
   A: To distinguish instance variables from parameters, call current class methods, constructor chaining, pass current object as parameter, return current object, and pass as argument in constructor calls.

3. **Q:** Can we use `this` in static methods?
   A: No, because `this` refers to an instance, and static methods don't belong to any instance.

4. **Q:** What is the difference between `this` and `super`?

A: `this` refers to the current class object, while `super` refers to the parent class object.

5. **Q:** Can we use both `this()` and `super()` in the same constructor?
   A: No, because both must be the first statement, and only one statement can be first.

6. **Q:** Why use `this` for method chaining?
   A: Returning `this` from a method allows calling another method on the same object in a chain, making code more fluent and readable.

Chapter 9: Access Modifiers

Student–Expert Dialogue

Student: I've seen keywords like `public`, `private`, and `protected` in code. What are these?

Expert: These are called **access modifiers**. They control the visibility and accessibility of classes, methods, and variables. Think of them as security levels for your code.

```
1 class Demo {
2     public int publicVar = 1;         // Accessible everywhere
3     private int privateVar = 2;       // Accessible only in this
  class
4     protected int protectedVar = 3;   // Accessible in this class
  and subclasses
5     int defaultVar = 4;               // Accessible in same
  package (no keyword)
6 }
```

Student: Why do we need different access levels?

Expert: For **encapsulation** and **security**. You want to: 1. Hide implementation details 2. Prevent unauthorized access 3. Control how your code is used 4. Make changes without breaking other code

Let me show you with an example:

```java
// BAD: Everything is public
class BankAccount_Bad {
    public double balance;  // Anyone can modify!

    public void withdraw(double amount) {
        balance -= amount;
    }
}
// GOOD: Proper encapsulation
class BankAccount_Good {
    private double balance;  // Hidden from outside

    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        } else {
            System.out.println("Invalid withdrawal!");
        }
    }

    public double getBalance() {
        return balance;
    }
}
public class Main {
    public static void main(String[] args) {
        BankAccount_Bad bad = new BankAccount_Bad();
        bad.balance = 1000;
        bad.balance = -5000;  // ❌ Can set negative balance!

        BankAccount_Good good = new BankAccount_Good();
        // good.balance = 1000;  // ❌ Compile error - cannot access private field
        good.withdraw(-5000);  // ✅ Validation prevents this
    }
}
```

The Four Access Modifiers

Expert: Java has four access modifiers. Let me explain each:

1. Public

```java
// Accessible from ANYWHERE
public class PublicDemo {
    public int publicVar = 10;

    public void publicMethod() {
        System.out.println("Public method - accessible everywhere");
    }
}
// Can be accessed from any class, any package
class AnyClass {
    void test() {
        PublicDemo demo = new PublicDemo();
        System.out.println(demo.publicVar);  // ✅ OK
        demo.publicMethod();  // ✅ OK
    }
}
```

2. Private

```java
// Accessible ONLY within the same class
class PrivateDemo {
    private int privateVar = 20;

    private void privateMethod() {
        System.out.println("Private method - only accessible in this class");
    }

    public void publicMethod() {
        // Can access private members within same class
        System.out.println(privateVar);  // ✅ OK
        privateMethod();  // ✅ OK
    }
}
class AnotherClass {
    void test() {
        PrivateDemo demo = new PrivateDemo();
        // System.out.println(demo.privateVar);  // ❌ Compile error
        // demo.privateMethod();  // ❌ Compile error
        demo.publicMethod();  // ✅ OK
    }
}
```

3. Protected

```java
// Accessible within same package and subclasses
class ProtectedDemo {
    protected int protectedVar = 30;

    protected void protectedMethod() {
        System.out.println("Protected method");
    }
}
// Same package - can access
class SamePackageClass {
    void test() {
        ProtectedDemo demo = new ProtectedDemo();
        System.out.println(demo.protectedVar);  // ✅ OK (same package)
        demo.protectedMethod();  // ✅ OK (same package)
    }
}
// Different package but subclass - can access
class SubclassInDifferentPackage extends ProtectedDemo {
    void test() {
        System.out.println(protectedVar);  // ✅ OK (inherited)
        protectedMethod();  // ✅ OK (inherited)
    }
}
// Different package, not subclass - cannot access
class DifferentPackageClass {
    void test() {
        ProtectedDemo demo = new ProtectedDemo();
        // System.out.println(demo.protectedVar);  // ❌ Compile error
        // demo.protectedMethod();  // ❌ Compile error
    }
}
```

4. Default (Package-Private)

```java
// Accessible only within the same package (no keyword)
class DefaultDemo {
    int defaultVar = 40;  // No access modifier = default

    void defaultMethod() {
        System.out.println("Default method");
    }
}
// Same package - can access
class SamePackageClass {
    void test() {
        DefaultDemo demo = new DefaultDemo();
        System.out.println(demo.defaultVar);  // ✅ OK
        demo.defaultMethod();  // ✅ OK
    }
}
// Different package - cannot access
/*
package other;
class DifferentPackageClass {
    void test() {
        DefaultDemo demo = new DefaultDemo();
        // System.out.println(demo.defaultVar);  // ❌ Compile error
        // demo.defaultMethod();  // ❌ Compile error
    }
}
*/
```

Access Modifier Comparison Table

Student: Can you show me a comparison table?

Expert: Absolutely! Here's the complete picture:

Access Level Table:

Modifier	Class	Package	Subclass	World
public	✅	✅	✅	✅
protected	✅	✅	✅	❌
default	✅	✅	❌	❌
private	✅	❌	❌	❌

Legend:
- Class: Within the same class
- Package: Within the same package
- Subclass: In a subclass (different package)
- World: Everywhere (different package, not subclass)

Complete Example with All Modifiers

Student: Can you show all modifiers in one example?

Expert: Sure! Let's create a comprehensive example:

```
class AccessModifierDemo {
    // Different access levels
    public int publicVar = 1;
    protected int protectedVar = 2;
    int defaultVar = 3;  // package-private
    private int privateVar = 4;

    // Public method - accessible everywhere
    public void publicMethod() {
        System.out.println("Public method called");
        // Can access all variables within same class
        System.out.println("Public: " + publicVar);
        System.out.println("Protected: " + protectedVar);
        System.out.println("Default: " + defaultVar);
        System.out.println("Private: " + privateVar);
    }

    // Protected method - accessible in package and subclasses
    protected void protectedMethod() {
        System.out.println("Protected method called");
```

```java
    }

    // Default method - accessible only in same package
    void defaultMethod() {
        System.out.println("Default method called");
    }

    // Private method - accessible only in this class
    private void privateMethod() {
        System.out.println("Private method called");
    }

    // Public method to demonstrate private method access
    public void callPrivateMethod() {
        privateMethod();  // Can call private method within same class
    }
}

// Same package class
class SamePackageTest {
    void test() {
        AccessModifierDemo demo = new AccessModifierDemo();

        System.out.println("=== Same Package Access ===");
        System.out.println("Public var: " + demo.publicVar); // ✅ OK
        System.out.println("Protected var: " + demo.protectedVar);  // ✅ OK
        System.out.println("Default var: " + demo.defaultVar); // ✅ OK
        // System.out.println("Private var: " + demo.privateVar); // ❌ Error

        demo.publicMethod();     // ✅ OK
        demo.protectedMethod();  // ✅ OK
        demo.defaultMethod();    // ✅ OK
        // demo.privateMethod();  // ❌ Error
        demo.callPrivateMethod();  // ✅ OK (calls private method indirectly)
    }
}

// Subclass in same package
class SubclassInSamePackage extends AccessModifierDemo {
    void test() {
        System.out.println("=== Subclass in Same Package ===");
        System.out.println("Public var: " + publicVar);      // ✅ OK (inherited)
        System.out.println("Protected var: " + protectedVar);  //
```

```
    ✅ OK (inherited)
        System.out.println("Default var: " + defaultVar);    //
    ✅ OK (inherited)
        // System.out.println("Private var: " + privateVar);  //
    ❌ Error (not inherited)

        publicMethod();     // ✅ OK
        protectedMethod();  // ✅ OK
        defaultMethod();    // ✅ OK
        // privateMethod();  // ❌ Error
    }
}
public class Main {
    public static void main(String[] args) {
        SamePackageTest test1 = new SamePackageTest();
        test1.test();

        System.out.println();

        SubclassInSamePackage test2 = new
    SubclassInSamePackage();
        test2.test();
    }
}
```

Real-World Example: Employee Management System

Student: Can you show a real-world example with proper access modifiers?

Expert: Absolutely! Let's build an employee management system:

```
class Employee {
    // Private fields - hidden from outside
    private String employeeId;
    private String name;
    private double salary;
    private String department;

    // Protected field - accessible to subclasses
    protected int yearsOfExperience;

    // Package-private field - accessible in same package
    String officeLocation;

    // Public constructor
```

```
15      public Employee(String employeeId, String name, double
    salary, String department) {
16          this.employeeId = employeeId;
17          this.name = name;
18          this.salary = salary;
19          this.department = department;
20          this.yearsOfExperience = 0;
21          this.officeLocation = "Head Office";
22      }
23
24      // Public getters - controlled read access
25      public String getEmployeeId() {
26          return employeeId;
27      }
28
29      public String getName() {
30          return name;
31      }
32
33      public String getDepartment() {
34          return department;
35      }
36
37      // Public setter with validation
38      public void setName(String name) {
39          if (name != null && !name.trim().isEmpty()) {
40              this.name = name;
41          }
42      }
43
44      // Protected method - accessible to subclasses
45      protected double getSalary() {
46          return salary;
47      }
48
49      // Protected method with validation
50      protected void setSalary(double salary) {
51          if (salary > 0) {
52              this.salary = salary;
53          }
54      }
55
56      // Private helper method - internal use only
57      private double calculateTax() {
58          return salary * 0.2;  // 20% tax
59      }
60
61      // Private helper method
```

```java
    private double calculateBonus() {
        return salary * 0.1 * yearsOfExperience;
    }

    // Public method using private methods
    public double getNetSalary() {
        return salary - calculateTax() + calculateBonus();
    }

    // Package-private method
    void updateOfficeLocation(String location) {
        this.officeLocation = location;
    }

    // Public method
    public void displayInfo() {
        System.out.println("=== Employee Information ===");
        System.out.println("ID: " + employeeId);
        System.out.println("Name: " + name);
        System.out.println("Department: " + department);
        System.out.println("Office: " + officeLocation);
        System.out.println("Experience: " + yearsOfExperience + "
years");
        System.out.println("Gross Salary: $" + salary);
        System.out.println("Tax: $" + calculateTax());
        System.out.println("Bonus: $" + calculateBonus());
        System.out.println("Net Salary: $" + getNetSalary());
    }
}
class Manager extends Employee {
    private int teamSize;

    public Manager(String employeeId, String name, double salary,
String department, int teamSize) {
        super(employeeId, name, salary, department);
        this.teamSize = teamSize;
    }

    // Can access protected members from parent
    public void giveRaise(double percentage) {
        double currentSalary = getSalary();  // ✅ OK -
protected method
        double newSalary = currentSalary * (1 + percentage /
100);
        setSalary(newSalary);  // ✅ OK - protected method
        System.out.println("Raise given! New salary: $" +
newSalary);
    }
```

```java

    public void addExperience(int years) {
        yearsOfExperience += years;  // ✅ OK - protected field
    }

    @Override
    public void displayInfo() {
        super.displayInfo();
        System.out.println("Team Size: " + teamSize);
    }
}
// Same package class - HR Department
class HRDepartment {
    public void processEmployee(Employee emp) {
        System.out.println("=== HR Processing ===");
        System.out.println("Employee: " + emp.getName());  // ✅ OK - public
        System.out.println("Department: " + emp.getDepartment());  // ✅ OK - public
        System.out.println("Office: " + emp.officeLocation);  // ✅ OK - package-private

        // Can update office location (package-private method)
        emp.updateOfficeLocation("Branch Office");  // ✅ OK

        // Cannot access private or protected members
        // System.out.println("Salary: " + emp.getSalary());  // ❌ Error - protected
        // System.out.println("Salary: " + emp.salary);  // ❌ Error - private
    }
}
public class Main {
    public static void main(String[] args) {
        Employee emp = new Employee("E001", "Alice", 50000, "IT");
        emp.yearsOfExperience = 5;  // ✅ OK - protected (same package)
        emp.displayInfo();

        System.out.println();

        Manager mgr = new Manager("M001", "Bob", 80000, "Management", 10);
        mgr.addExperience(10);
        mgr.giveRaise(10);
        mgr.displayInfo();

```

```
148         System.out.println();
149
150         HRDepartment hr = new HRDepartment();
151         hr.processEmployee(emp);
152
153         System.out.println();
154         emp.displayInfo();
155     }
156 }
```

Output:

```
=== Employee Information ===
ID: E001
Name: Alice
Department: IT
Office: Head Office
Experience: 5 years
Gross Salary: $50000.0
Tax: $10000.0
Bonus: $25000.0
Net Salary: $65000.0

Raise given! New salary: $88000.0
=== Employee Information ===
ID: M001
Name: Bob
Department: Management
Office: Head Office
Experience: 10 years
Gross Salary: $88000.0
Tax: $17600.0
Bonus: $88000.0
Net Salary: $158400.0
Team Size: 10

=== HR Processing ===
Employee: Alice
Department: IT
Office: Head Office

=== Employee Information ===
ID: E001
Name: Alice
Department: IT
Office: Branch Office
Experience: 5 years
Gross Salary: $50000.0
Tax: $10000.0
Bonus: $25000.0
Net Salary: $65000.0
```

Access Modifiers for Classes

Student: Can classes also have access modifiers?

Expert: Yes, but with restrictions:

```java
// ✅ Public class - accessible from anywhere
// File name MUST match class name (PublicClass.java)
public class PublicClass {
    public void display() {
        System.out.println("Public class");
    }
}
// ✅ Default class - accessible only in same package
// Can have any file name
class DefaultClass {
    void display() {
        System.out.println("Default class");
    }
}
// ❌ Cannot have protected or private top-level classes
// protected class ProtectedClass { }  // Compile error
// private class PrivateClass { }  // Compile error
// ✅ But can have private/protected nested classes
class OuterClass {
    private class PrivateInnerClass {
        void display() {
            System.out.println("Private inner class");
        }
    }

    protected class ProtectedInnerClass {
        void display() {
            System.out.println("Protected inner class");
        }
    }

    public void test() {
        PrivateInnerClass pic = new PrivateInnerClass();
        pic.display();  // ✅ OK - same outer class
    }
}
```

Rules for Class Access Modifiers:

```
Top-level Classes:
┌──────────────────────────────────────┐
│ ✅ public - accessible everywhere    │
│ ✅ default - accessible in same package│
│ ❌ protected - NOT allowed           │
│ ❌ private - NOT allowed             │
└──────────────────────────────────────┘

Nested/Inner Classes:
┌──────────────────────────────────────┐
│ ✅ public - accessible everywhere    │
│ ✅ protected - accessible in subclasses│
│ ✅ default - accessible in same package│
│ ✅ private - accessible in outer class │
└──────────────────────────────────────┘
```

Best Practices for Access Modifiers

Student: How do I decide which access modifier to use?

Expert: Follow these guidelines:

```java
class BestPracticesDemo {
    // ✅ RULE 1: Make fields private by default
    private String name;
    private int age;
    private double salary;

    // ✅ RULE 2: Provide public getters/setters only when needed
    public String getName() {
        return name;
    }

    public void setName(String name) {
        if (name != null && !name.trim().isEmpty()) {
            this.name = name;
        }
    }

    // ✅ RULE 3: Read-only fields - getter but no setter
    public int getAge() {
```

```java
        return age;
    }

    // ✅ RULE 4: No getter for sensitive data
    // No getSalary() method - salary is completely hidden

    // ✅ RULE 5: Use protected for inheritance-specific members
    protected void calculateBonus() {
        // Subclasses can override this
    }

    // ✅ RULE 6: Private helper methods
    private boolean isValidAge(int age) {
        return age >= 18 && age <= 65;
    }

    // ✅ RULE 7: Public API methods
    public void updateAge(int newAge) {
        if (isValidAge(newAge)) {
            this.age = newAge;
        }
    }
}
// ❌ BAD PRACTICE
class BadPractice {
    public String name;  // Anyone can modify
    public int age;      // No validation
    public double salary;  // Sensitive data exposed

    // No encapsulation, no control
}
// ✅ GOOD PRACTICE
class GoodPractice {
    private String name;
    private int age;
    private double salary;

    // Controlled access with validation
    public void setAge(int age) {
        if (age >= 0 && age <= 150) {
            this.age = age;
        } else {
            throw new IllegalArgumentException("Invalid age");
        }
    }

    public int getAge() {
        return age;
```

```
70     }
71 }
```

Decision Tree:

```
Choosing Access Modifier:
┌─────────────────────────────────────────┐
│ Should everyone access it?              │
│   YES → public                          │
│   NO ↓                                  │
│                                         │
│ Should subclasses access it?            │
│   YES → protected                       │
│   NO ↓                                  │
│                                         │
│ Should same package access it?          │
│   YES → default (no modifier)           │
│   NO ↓                                  │
│                                         │
│ Only this class needs it?               │
│   YES → private                         │
└─────────────────────────────────────────┘

General Rule: Start with private,
              increase access only when needed
```

Common Scenarios

Student: Can you show common real-world scenarios?

Expert: Sure! Here are typical use cases:

Scenario 1: Data Hiding (Encapsulation)

```java
class CreditCard {
    // Private - sensitive data
    private String cardNumber;
    private String cvv;
    private String pin;

    // Public - safe operations
    public boolean validatePin(String enteredPin) {
        return this.pin.equals(enteredPin);
    }

    // Public - masked display
    public String getMaskedCardNumber() {
        return "****-****-****-" + cardNumber.substring(12);
    }

    // Private - internal validation
    private boolean isValidCardNumber(String number) {
        return number.length() == 16 && number.matches("\\d+");
    }
}
```

Scenario 2: Template Method Pattern

```java
abstract class DataProcessor {
    // Public - template method
    public final void process() {
        loadData();
        validateData();
        processData();
        saveData();
    }

    // Protected - subclasses can override
    protected abstract void loadData();
    protected abstract void processData();

    // Private - cannot be overridden
    private void validateData() {
        System.out.println("Validating data...");
    }

    // Protected - subclasses can override
    protected void saveData() {
        System.out.println("Saving data...");
    }
}
class CSVProcessor extends DataProcessor {
    @Override
    protected void loadData() {
        System.out.println("Loading CSV data");
    }

    @Override
    protected void processData() {
        System.out.println("Processing CSV data");
    }
}
```

Scenario 3: Builder Pattern

```java
class User {
    // Private fields
    private final String username;  // Required
    private final String email;     // Required
    private String firstName;       // Optional
    private String lastName;        // Optional
    private int age;                // Optional
```

```
8
9       // Private constructor
10      private User(UserBuilder builder) {
11          this.username = builder.username;
12          this.email = builder.email;
13          this.firstName = builder.firstName;
14          this.lastName = builder.lastName;
15          this.age = builder.age;
16      }
17
18      // Public static nested builder class
19      public static class UserBuilder {
20          // Required parameters
21          private final String username;
22          private final String email;
23
24          // Optional parameters
25          private String firstName = "";
26          private String lastName = "";
27          private int age = 0;
28
29          // Public constructor with required parameters
30          public UserBuilder(String username, String email) {
31              this.username = username;
32              this.email = email;
33          }
34
35          // Public setter methods that return builder
36          public UserBuilder firstName(String firstName) {
37              this.firstName = firstName;
38              return this;
39          }
40
41          public UserBuilder lastName(String lastName) {
42              this.lastName = lastName;
43              return this;
44          }
45
46          public UserBuilder age(int age) {
47              this.age = age;
48              return this;
49          }
50
51          // Public build method
52          public User build() {
53              return new User(this);
54          }
55      }
```

```
56
57      // Public getters
58      public String getUsername() {
59          return username;
60      }
61
62      public String getEmail() {
63          return email;
64      }
65
66      @Override
67      public String toString() {
68          return "User{username='" + username + "', email='" +
    email +
69                  "', firstName='" + firstName + "', lastName='" +
    lastName +
70                  "', age=" + age + "}";
71      }
72  }
73  public class Main {
75      public static void main(String[] args) {
76          // Cannot create User directly - constructor is private
77          // User user = new User();  // ❌ Compile error
78
79          // Must use builder
80          User user = new User.UserBuilder("john_doe",
    "john@example.com")
81                              .firstName("John")
82                              .lastName("Doe")
83                              .age(30)
84                              .build();
85
86          System.out.println(user);
87      }
88  }
```

Interview Trap Alert! 🚨

Interviewer might ask: “What is the default access modifier in Java?”

Wrong Answer: “There is no default access modifier.”

Correct Answer: “When no access modifier is specified, it’s called ‘default’ or ‘package-private’ access. Members are accessible only within the same package.”

```java
// No access modifier = default/package-private
class DefaultExample {
    int value;  // default access

    void display() {  // default access
        System.out.println("Default access");
    }
}
```

Another Common Question: “Can we reduce the access level when overriding a method?”

Wrong Answer: “Yes, we can change access level to anything.”

Correct Answer: “No, we cannot reduce the access level. We can only keep it the same or increase it. This is because of the Liskov Substitution Principle - a subclass should be usable wherever the parent class is used.”

```java
class Parent {
    protected void display() {
        System.out.println("Parent");
    }
}
class Child extends Parent {
    // ✅ OK - same access level
    @Override
    protected void display() {
        System.out.println("Child");
    }

    // ✅ OK - increased access level
    /*
    @Override
    public void display() {
        System.out.println("Child");
    }
    */

    // ❌ ERROR - reduced access level
    /*
    @Override
    private void display() {
        System.out.println("Child");
    }
    */
}
```

Access Level Hierarchy:

```
More Restrictive → Less Restrictive
private < default < protected < public

When overriding:
✅ Can move right (increase access)
❌ Cannot move left (reduce access)
```

Third Common Question: "Can a private method be overridden?"

Answer: "No, private methods cannot be overridden because they are not inherited. If a subclass has a method with the same signature, it's a completely new method, not an

override.”

```java
class Parent {
    private void display() {
        System.out.println("Parent private");
    }

    public void callDisplay() {
        display();  // Calls Parent's private method
    }
}
class Child extends Parent {
    // This is NOT overriding - it's a new method
    private void display() {
        System.out.println("Child private");
    }

    public void callChildDisplay() {
        display();  // Calls Child's own method
    }
}
public class Main {
    public static void main(String[] args) {
        Child child = new Child();
        child.callDisplay();       // Parent private
        child.callChildDisplay();  // Child private
    }
}
```

Chapter Recap

✅ Four access modifiers: public, protected, default, private
✅ public = accessible everywhere
✅ protected = accessible in same package and subclasses
✅ default = accessible only in same package
✅ private = accessible only in same class
✅ Use private for fields, public for necessary methods
✅ Cannot reduce access level when overriding
✅ Top-level classes can only be public or default

Interview Questions

1. **Q:** What are access modifiers in Java?
   A: Keywords that set the accessibility/visibility of classes, methods, and variables. Java has four: public, protected, default, and private.

2. **Q:** What is the difference between public and private?
   A: Public members are accessible from anywhere, while private members are accessible only within the same class.

3. **Q:** What is the default access modifier?
   A: When no modifier is specified, it's package-private (default), accessible only within the same package.

4. **Q:** Can we override a private method?
   A: No, private methods are not inherited, so they cannot be overridden.

5. **Q:** Can we reduce the access level when overriding?
   A: No, we can only keep it the same or increase it (e.g., protected to public is OK, but public to protected is not).

6. **Q:** What access modifiers can be used for top-level classes?
   A: Only public and default. Protected and private are not allowed for top-level classes.

7. **Q:** Why make variables private?
   A: To implement encapsulation, hide implementation details, control access through methods, and add validation.

8. **Q:** What is the most restrictive access modifier?
   A: private - accessible only within the same class.

This completes **Book 1: Basic Java OOP Concepts**!

The book covers all fundamental OOP concepts that freshers need to know: - What is OOP and why Java - Classes and Objects - Encapsulation - Abstraction - Inheritance - Polymorphism - Constructors - The `this` keyword - Access Modifiers

Each chapter includes: ✅ Student-Expert dialogue format ✅ Code examples with explanations ✅ Real-world scenarios ✅ Interview traps and common mistakes ✅ Chapter recap ✅ Interview questions