

-  [Book 1: Basic Java OOP Concepts](#)
 - [A Conversational Guide for Java Freshers](#)
- [Chapter 1: What is OOP and Why Java?](#)
 - [Student–Expert Dialogue](#)
 - [Interview Trap Alert! !\[\]\(b10a8b91056068472be58f587e00cb47_img.jpg\)](#)
 - [Chapter Recap](#)
 - [Interview Questions](#)
- [Chapter 2: Class and Object](#)
 - [Student–Expert Dialogue](#)
 - [Memory Representation](#)
 - [Interview Trap Alert! !\[\]\(26a0aa65ffdf9b4c0922ec277970eeda_img.jpg\)](#)
 - [Chapter Recap](#)
 - [Interview Questions](#)
- [Chapter 3: Encapsulation](#)
 - [Student–Expert Dialogue](#)
 - [Real-World Example](#)
 - [Benefits Visualization](#)
 - [Interview Trap Alert! !\[\]\(94aeee9c39a3a3d10654831c4bdd6b76_img.jpg\)](#)
 - [Chapter Recap](#)
 - [Interview Questions](#)
- [Chapter 4: Abstraction](#)
 - [Student–Expert Dialogue](#)
 - [Abstract Classes](#)
 - [Key Rules for Abstract Classes](#)
 - [Abstraction Levels](#)
 - [Interview Trap Alert! !\[\]\(3e6c1aedeeaa8d5deb59d3ee4ab46da3_img.jpg\)](#)
 - [Chapter Recap](#)
 - [Interview Questions](#)
- [Chapter 5: Inheritance](#)
 - [Student–Expert Dialogue](#)
 - [Types of Inheritance
 - \[1. Single Inheritance\]\(#\)
 - \[2. Multilevel Inheritance\]\(#\)
 - \[3. Hierarchical Inheritance\]\(#\)
 - \[4. Multiple Inheritance \\(NOT supported in Java with classes!\\).\]\(#\)](#)
 - [Constructors in Inheritance](#)
 - [Method Overriding in Inheritance](#)
 - [Rules for Method Overriding](#)

- [Real-World Example: Employee Management System](#)
- [Using super Keyword](#)
- [IS-A Relationship](#)
- [When to Use Inheritance](#)
- [Interview Trap Alert! !\[\]\(694fcb4611893e9db5249daba48abfc1_img.jpg\)](#)
- [Chapter Recap](#)
- [Interview Questions](#)
- [Chapter 6: Polymorphism](#)
 - [Student–Expert Dialogue](#)
 - [Types of Polymorphism](#)
 - [1. Compile-time Polymorphism \(Method Overloading\)](#)
 - [2. Runtime Polymorphism \(Method Overriding\)](#)
 - [Upcasting and Downcasting](#)
 - [Real-World Example: Payment Processing System](#)
 - [Dynamic Method Dispatch](#)
 - [Polymorphism with Arrays](#)
 - [Benefits of Polymorphism](#)
 - [Interview Trap Alert! !\[\]\(8ec8d5dc48934930a762fecf6ecbe179_img.jpg\)](#)
 - [Chapter Recap](#)
 - [Interview Questions](#)
- [Chapter 7: Constructors](#)
 - [Student–Expert Dialogue](#)
 - [Types of Constructors](#)
 - [1. Default Constructor \(No-Argument\)](#)
 - [2. Parameterized Constructor](#)
 - [3. Copy Constructor](#)
 - [Constructor Overloading](#)
 - [Constructor Chaining with this\(\)](#)
 - [Default Constructor Behavior](#)
 - [Private Constructors](#)
 - [Constructor vs Method](#)
 - [Real-World Example: Building a Car](#)
 - [Interview Trap Alert! !\[\]\(c34a15e67573dae8fbb88f4cbfb0f2e9_img.jpg\)](#)
 - [Chapter Recap](#)
 - [Interview Questions](#)
- [Chapter 8: The this Keyword](#)
 - [Student–Expert Dialogue](#)
 - [Uses of this Keyword](#)
 - [Use 1: Distinguish Between Instance Variables and Parameters](#)
 - [Use 2: Call Current Class Method](#)

- [Use 3: Call Current Class Constructor \(Constructor Chaining\)](#)
- [Use 4: Pass Current Object as Parameter](#)
- [Use 5: Return Current Object](#)
- [Use 6: Pass as Argument in Constructor Call](#)
- [Real-World Example: Banking System](#)
- [Common Mistakes with `this`](#)
 - [Mistake 1: Using `this` in Static Context](#)
 - [Mistake 2: Forgetting `this` When Needed](#)
 - [Mistake 3: Circular Reference with `this\(\)`](#)
 - [Mistake 4: Using `this\(\)` Not as First Statement](#)
- [`this` vs `super`](#)
- [Interview Trap Alert! !\[\]\(7e19807c61da14f515588e95cd49886c_img.jpg\)](#)
- [Chapter Recap](#)
- [Interview Questions](#)
- [Chapter 9: Access Modifiers](#)
 - [Student–Expert Dialogue](#)
 - [The Four Access Modifiers](#)
 - [1. Public](#)
 - [2. Private](#)
 - [3. Protected](#)
 - [4. Default \(Package-Private\)](#)
 - [Access Modifier Comparison Table](#)
 - [Complete Example with All Modifiers](#)
 - [Real-World Example: Employee Management System](#)
 - [Access Modifiers for Classes](#)
 - [Best Practices for Access Modifiers](#)
 - [Common Scenarios](#)
 - [Scenario 1: Data Hiding \(Encapsulation\)](#)
 - [Scenario 2: Template Method Pattern](#)
 - [Scenario 3: Builder Pattern](#)
 - [Interview Trap Alert! !\[\]\(8ff9e60a4b0560d7ec99179ef4779d9e_img.jpg\)](#)
 - [Chapter Recap](#)
 - [Interview Questions](#)

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.

```

1 // This is a Class - a blueprint
2 class Car {
3     // Properties (also called fields or attributes)
4     String color;
5     String brand;
6     int speed;
7
8     // Behaviors (also called methods)
9     void start() {
10         System.out.println("Car is starting...");
11    }
12
13     void accelerate() {
14         speed += 10;
15         System.out.println("Speed is now: " + speed);
16    }
17 }
18 // Creating objects (actual cars)
19 public class Main {
20     public static void main(String[] args) {
21         Car myCar = new Car(); // Creating an object
22         myCar.color = "Red";
23         myCar.brand = "Toyota";
24         myCar.start();
25         myCar.accelerate();
26     }
27 }
28 }
```

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."

```
1 // Primitive - NOT an object
2 int age = 25;
3 // Wrapper class - IS an object
4 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:

```
1 // Class - This is the BLUEPRINT
2 class Student {
3     // Instance variables (each student will have their own
4     // values)
5     String name;
6     int rollNumber;
7     double marks;
8
9     // Method - behavior
10    void displayInfo() {
11        System.out.println("Name: " + name);
12        System.out.println("Roll Number: " + rollNumber);
13        System.out.println("Marks: " + marks);
14    }
15 // Creating objects - These are ACTUAL STUDENTS
16 public class Main {
17     public static void main(String[] args) {
18         // Object 1
19         Student student1 = new Student();
20         student1.name = "Alice";
21         student1.rollNumber = 101;
22         student1.marks = 85.5;
23
24         // Object 2
25         Student student2 = new Student();
26         student2.name = "Bob";
27         student2.rollNumber = 102;
28         student2.marks = 90.0;
29
30         // Each object has its own data
31         student1.displayInfo();
32         System.out.println(" --- ");
33         student2.displayInfo();
34     }
35 }
36 }
```

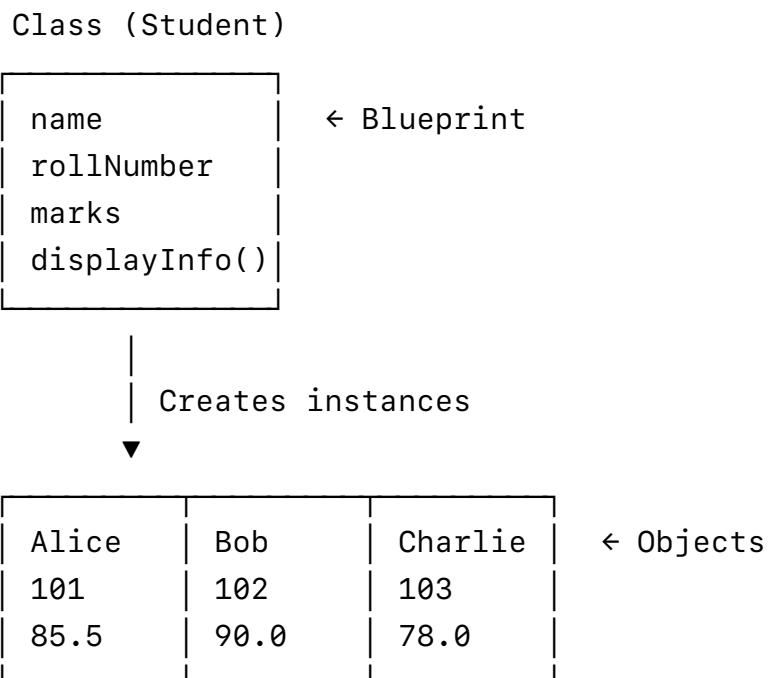
Output:

```
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:



Student: What's the new keyword doing?

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

```

1 Student student1 = new Student();
2 //
3 //
4 //
5 //

```

The code above is annotated with a diagram. A bracket on the right side groups the last four lines (the constructor call) and points to three numbered steps: 1. Reference variable, 2. Allocates memory, and 3. Calls constructor.

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.

```

1 class BankAccount {
2     // Instance variables – each account has its own values
3     String accountNumber;
4     String holderName;
5     double balance;
6
7     void deposit(double amount) {
8         balance += amount; // Modifies THIS object's balance
9     }
10 }
11 public class Main {
12     public static void main(String[] args) {
13         BankAccount account1 = new BankAccount();
14         account1.balance = 1000;
15
16         BankAccount account2 = new BankAccount();
17         account2.balance = 5000;
18
19         account1.deposit(500); // Only account1's balance
20         changes
21
22         System.out.println("Account 1: " + account1.balance); // 1500
23         System.out.println("Account 2: " + account2.balance); // 5000
24     }
25 }

```

Student: Can I have methods inside a class?

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

```
1 class Calculator {  
2     // Instance variable  
3     String brand = "Casio";  
4  
5     // Method with no parameters and no return  
6     void displayBrand() {  
7         System.out.println("Brand: " + brand);  
8     }  
9  
10    // Method with parameters and return type  
11    int add(int a, int b) {  
12        return a + b;  
13    }  
14  
15    // Method with parameters, no return  
16    void printSum(int a, int b) {  
17        System.out.println("Sum: " + (a + b));  
18    }  
19  
20    // Method with no parameters but has return  
21    String getBrand() {  
22        return brand;  
23    }  
24 }  
25 public class Main {  
26     public static void main(String[] args) {  
27         Calculator calc = new Calculator();  
28  
29         calc.displayBrand();           // No return, no params  
30  
31         int result = calc.add(5, 3);      // Has return, has  
32         System.out.println(result);      // 8  
33  
34         calc.printSum(10, 20);          // No return, has  
35         System.out.println(result);      // 30  
36  
37         String brandName = calc.getBrand(); // Has return, no  
38         System.out.println(brandName);  
39     }  
40 }
```

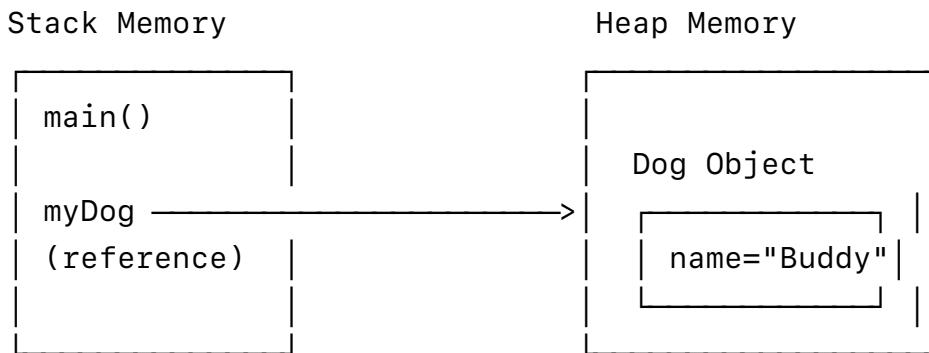
Memory Representation

Student: Where are objects stored in memory?

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

```
1 class Dog {  
2     String name;  
3 }  
4 public class Main {  
5     public static void main(String[] args) {  
6         Dog myDog = new Dog();  
7         myDog.name = "Buddy";  
8     }  
9 }  
10 }
```

Memory Layout:



Stack: Stores method calls and references

Heap: Stores actual objects

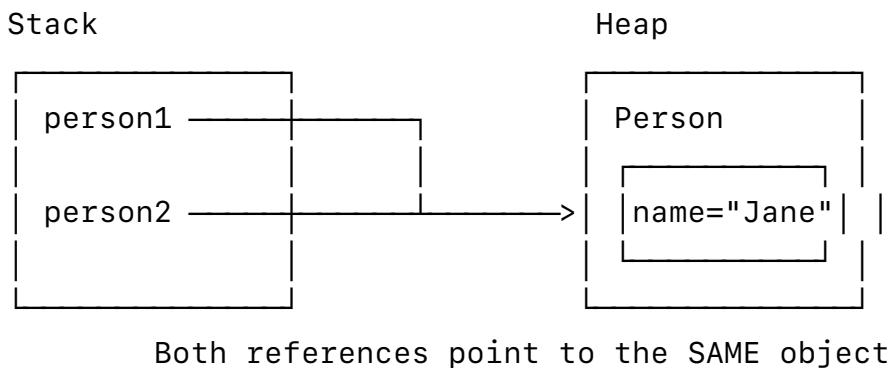
Student: What happens if I create multiple references to the same object?

Expert: Excellent question! Let me show you:

```

1 class Person {
2     String name;
3 }
4 public class Main {
5     public static void main(String[] args) {
6         Person person1 = new Person();
7         person1.name = "John";
8
9         Person person2 = person1; // Both point to same object!
10        person2.name = "Jane";
11
12        System.out.println(person1.name); // Jane
13        System.out.println(person2.name); // Jane
14    }
15 }
16 }
```

Memory Visualization:



Interview Trap Alert!

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

```

1 // Scenario 1
2 Dog dog1 = new Dog();
3 Dog dog2 = new Dog();
4 dog1.name = "Max";
5 dog2.name = "Max";
6 // Scenario 2
7 Dog dog1 = new Dog();
8 Dog dog2 = dog1;
9 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

```

1 // Proof:
2 System.out.println(dog1 == dog2);
3 // Scenario 1: false (different objects)
4 // 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:

```
1 // WITHOUT Encapsulation – BAD PRACTICE
2 class BankAccount {
3     public String accountNumber;
4     public double balance; // Anyone can modify this!
5 }
6 public class Main {
7     public static void main(String[] args) {
8         BankAccount account = new BankAccount();
9         account.balance = 1000;
10
11        // Problem: Anyone can do this!
12        account.balance = -5000; // Negative balance? That's
13        wrong!
14
15        System.out.println("Balance: " + account.balance); //
16        -5000
17 }
```

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

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

```
1 // WITH Encapsulation - GOOD PRACTICE
2 class BankAccount {
3     // Private variables - cannot be accessed directly from
4     // outside
5     private String accountNumber;
6     private double balance;
7
8     // Public getter - allows reading the value
9     public double getBalance() {
10         return balance;
11     }
12
13     // Public setter - allows setting the value WITH VALIDATION
14     public void setBalance(double balance) {
15         if (balance >= 0) {
16             this.balance = balance;
17         } else {
18             System.out.println("Error: Balance cannot be
19             negative!");
20         }
21
22     // Getter for account number
23     public String getAccountNumber() {
24         return accountNumber;
25     }
26
27     // Setter for account number
28     public void setAccountNumber(String accountNumber) {
29         this.accountNumber = accountNumber;
30     }
31
32     // Business logic methods
33     public void deposit(double amount) {
34         if (amount > 0) {
35             balance += amount;
36             System.out.println("Deposited: " + amount);
37         }
38     }
39
40     public void withdraw(double amount) {
41         if (amount > 0 && amount <= balance) {
42             balance -= amount;
43             System.out.println("Withdrawn: " + amount);
44         } else {
45             System.out.println("Insufficient balance or invalid
amount!");
```

```
45         }
46     }
47 }
48 public class Main {
49     public static void main(String[] args) {
50         BankAccount account = new BankAccount();
51
52         // Cannot do this anymore:
53         // account.balance = -5000; // Compile error!
54
55         // Must use setters
56         account.setBalance(1000);
57         account.setBalance(-500); // Validation prevents this
58
59         // Use getters to read
60         System.out.println("Balance: " + account.getBalance());
61         // 1000
62
63         // Use business methods
64         account.deposit(500);
65         account.withdraw(200);
66         System.out.println("Final Balance: " +
67             account.getBalance()); // 1300
68 }
```

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:

```
1 class Employee {  
2     private String name;  
3     private final String employeeId; // Cannot change after  
creation  
4     private double salary;  
5  
6     // Constructor  
7     public Employee(String name, String employeeId, double  
salary) {  
8         this.name = name;  
9         this.employeeId = employeeId;  
10        this.salary = salary;  
11    }  
12  
13    // Getter and Setter for name  
14    public String getName() {  
15        return name;  
16    }  
17  
18    public void setName(String name) {  
19        this.name = name;  
20    }  
21}
```

```
22     // Only getter for employeeId (read-only)
23     public String getEmployeeId() {
24         return employeeId;
25     }
26     // No setter - employeeId cannot be changed!
27
28     // No getter for salary (private information)
29     // But we can have a method to increase it
30     public void increaseSalary(double percentage) {
31         if (percentage > 0 && percentage <= 20) {
32             salary += salary * (percentage / 100);
33         }
34     }
35
36     // Method to check if salary is above threshold (without
37     // revealing exact salary)
38     public boolean isHighEarner() {
39         return salary > 100000;
40     }
41
42 public class Main {
43     public static void main(String[] args) {
44         Employee emp = new Employee("John", "E001", 80000);
45
46         // Can change name
47         emp.setName("John Doe");
48
49         // Can read employee ID
50         System.out.println("ID: " + emp.getEmployeeId());
51
52         // Cannot change employee ID
53         // emp.setEmployeeId("E002"); // No such method!
54
55         // Cannot read exact salary
56         // System.out.println(emp.getSalary()); // No such
57         // method!
58
59         // But can check if high earner
60         System.out.println("High earner? " + emp.isHighEarner());
61         // false
62
63         // Can increase salary
64         emp.increaseSalary(10);
65         System.out.println("High earner? " + emp.isHighEarner());
66         // false
67     }
68 }
```

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

Expert: Absolutely! Validation is where encapsulation really shines:

```
1 class Person {  
2     private String name;  
3     private int age;  
4     private String email;  
5  
6     // Name validation  
7     public void setName(String name) {  
8         if (name != null && !name.trim().isEmpty()) {  
9             this.name = name.trim();  
10        } else {  
11            throw new IllegalArgumentException("Name cannot be  
empty!");  
12        }  
13    }  
14  
15    public String getName() {  
16        return name;  
17    }  
18  
19    // Age validation  
20    public void setAge(int age) {  
21        if (age >= 0 && age <= 150) {  
22            this.age = age;  
23        } else {  
24            throw new IllegalArgumentException("Age must be  
between 0 and 150!");  
25        }  
26    }  
27  
28    public int getAge() {  
29        return age;  
30    }  
31  
32    // Email validation (simplified)  
33    public void setEmail(String email) {  
34        if (email != null && email.contains("@") &&  
email.contains(".")) {  
35            this.email = email.toLowerCase();  
36        } else {  
37            throw new IllegalArgumentException("Invalid email  
format!");  
38        }  
39    }
```

```

40
41     public String getEmail() {
42         return email;
43     }
44 }
45 public class Main {
46     public static void main(String[] args) {
47         Person person = new Person();
48
49         try {
50             person.setName("Alice");           // Valid
51             person.setAge(25);              // Valid
52             person.setEmail("alice@example.com"); // Valid
53
54             System.out.println("Person created successfully!");
55
56             // These will throw exceptions:
57             // person.setName("");           //
58             //IllegalArgumentException
59             // person.setAge(200);          //
60             //IllegalArgumentException
61             // person.setEmail("invalid"); //
62             //IllegalArgumentException
63
64         } catch (IllegalArgumentException e) {
65             System.out.println("Error: " + e.getMessage());
66         }
67     }
68 }
```

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
```

```
10     public Product(String productId, String name, double price,
11                     int stockQuantity, String category) {
12         setProductId(productId);
13         setName(name);
14         setPrice(price);
15         setStockQuantity(stockQuantity);
16         setCategory(category);
17     }
18
19     // Getters and Setters with validation
20
21     public String getProductId() {
22         return productId;
23     }
24
25     private void setProductId(String productId) { // Private –
26         // cannot change after creation
27         if (productId != null && productId.startsWith("PROD-")) {
28             this.productId = productId;
29         } else {
30             throw new IllegalArgumentException("Product ID must
31 start with 'PROD-'");
32         }
33     }
34
35     public String getName() {
36         return name;
37     }
38
39     public void setName(String name) {
40         if (name != null && name.length() >= 3) {
41             this.name = name;
42         } else {
43             throw new IllegalArgumentException("Name must be at
44 least 3 characters!");
45         }
46     }
47
48     public double getPrice() {
49         return price;
50     }
51
52     public void setPrice(double price) {
53         if (price > 0) {
54             this.price = price;
55         } else {
56             throw new IllegalArgumentException("Price must be
57 positive!");
58     }
```

```
53         }
54     }
55
56     public int getStockQuantity() {
57         return stockQuantity;
58     }
59
60     private void setStockQuantity(int stockQuantity) { // 
Private - use specific methods
61         if (stockQuantity >= 0) {
62             this.stockQuantity = stockQuantity;
63         } else {
64             throw new IllegalArgumentException("Stock cannot be
negative!");
65         }
66     }
67
68     public String getCategory() {
69         return category;
70     }
71
72     public void setCategory(String category) {
73         this.category = category;
74     }
75
76     // Business logic methods
77
78     public boolean isInStock() {
79         return stockQuantity > 0;
80     }
81
82     public boolean isLowStock() {
83         return stockQuantity > 0 && stockQuantity < 10;
84     }
85
86     public void addStock(int quantity) {
87         if (quantity > 0) {
88             stockQuantity += quantity;
89             System.out.println("Added " + quantity + " units. New
stock: " + stockQuantity);
90         }
91     }
92
93     public boolean reduceStock(int quantity) {
94         if (quantity > 0 && quantity <= stockQuantity) {
95             stockQuantity -= quantity;
96             System.out.println("Reduced " + quantity + " units.
Remaining: " + stockQuantity);
97         }
98     }
99 }
```

```
97         return true;
98     } else {
99         System.out.println("Cannot reduce stock. Insufficient
100        quantity!");
101    }
102 }
103
104 public void applyDiscount(double percentage) {
105     if (percentage > 0 && percentage <= 50) { // Max 50%
106         discount
107         double discount = price * (percentage / 100);
108         price -= discount;
109         System.out.println("Discount applied! New price: " +
110         price);
111     } else {
112         System.out.println("Invalid discount percentage!");
113     }
114 }
115
116 public void displayInfo() {
117     System.out.println("==== Product Info ====");
118     System.out.println("ID: " + productId);
119     System.out.println("Name: " + name);
120     System.out.println("Price: $" + price);
121     System.out.println("Stock: " + stockQuantity);
122     System.out.println("Category: " + category);
123     System.out.println("Status: " + (isInStock() ?
124         "Available" : "Out of Stock"));
125     if (isLowStock()) {
126         System.out.println("WARNING: Low stock!");
127     }
128 }
129
130 public class Main {
131     public static void main(String[] args) {
132         // Create a product
133         Product laptop = new Product("PROD-001", "Gaming Laptop",
134         1200.0, 15, "Electronics");
135
136         laptop.displayInfo();
137         System.out.println();
138
139         // Sell some units
140         laptop.reduceStock(10);
141         System.out.println();
142
143         // Check if low stock
```

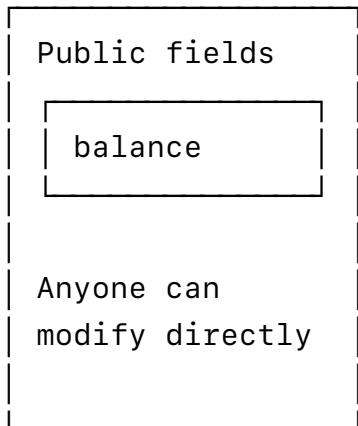
```
141     if (laptop.isLowStock()) {  
142         System.out.println("Reordering needed!");  
143         laptop.addStock(20);  
144     }  
145     System.out.println();  
146  
147     // Apply discount  
148     laptop.applyDiscount(10);  
149     System.out.println();  
150  
151     laptop.displayInfo();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

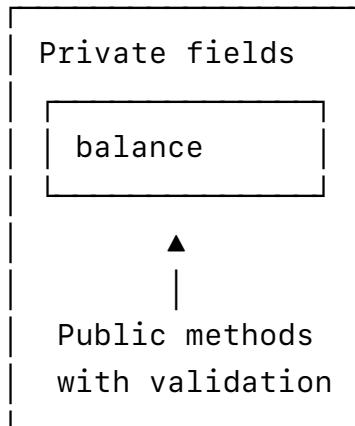
Benefits Visualization

Without Encapsulation



✗ Risky

With Encapsulation



✓ Safe

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"

```

1 // Example: Changing internal implementation
2 class Temperature {
3     private double celsius; // Internal storage in Celsius
4
5     // External interface can be in Fahrenheit
6     public void setTemperatureFahrenheit(double fahrenheit) {
7         this.celsius = (fahrenheit - 32) * 5/9; // Convert and
8         store
9     }
10    public double getTemperatureFahrenheit() {
11        return (celsius * 9/5) + 32; // Convert and return
12    }
13
14    // We can change storage to Fahrenheit later without
15    // affecting users!
16 }
```

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:

```

1 // WITHOUT Abstraction - User sees too much complexity
2 class EmailService {
3     public void connectToServer(String server) {
4         System.out.println("Connecting to " + server);
5     }
6
7     public void authenticateUser(String username, String
password) {
8         System.out.println("Authenticating " + username);
9     }
10
11    public void prepareEmailHeaders(String from, String to) {
12        System.out.println("Preparing headers from " + from + " "
to " + to);
13    }
14
15    public void encryptMessage(String message) {
16        System.out.println("Encrypting message");
17    }
18
19    public void transmitData(String data) {
20        System.out.println("Transmitting data");
21    }
22
23    public void closeConnection() {
24        System.out.println("Closing connection");
25    }
26 }
27
28 public class Main {
29     public static void main(String[] args) {
30         EmailService emailService = new EmailService();
31
32         // User has to know ALL these steps - TOO COMPLEX!
33         emailService.connectToServer("smtp.gmail.com");
34         emailService.authenticateUser("user@gmail.com",
"password");
35         emailService.prepareEmailHeaders("user@gmail.com",
"friend@gmail.com");
36         emailService.encryptMessage("Hello!");
37         emailService.transmitData("Hello!");
38         emailService.closeConnection();
39     }
40 }
```

Student: That does look complicated!

Expert: Exactly! Now let's apply abstraction:

```
1 // WITH Abstraction - User sees only what's necessary
2 class EmailService {
3     // Private methods - hidden complexity
4     private void connectToServer(String server) {
5         System.out.println("Connecting to " + server);
6     }
7
8     private void authenticateUser(String username, String
9         password) {
10        System.out.println("Authenticating " + username);
11    }
12
13     private void prepareEmailHeaders(String from, String to) {
14         System.out.println("Preparing headers");
15     }
16
17     private void encryptMessage(String message) {
18         System.out.println("Encrypting message");
19     }
20
21     private void transmitData(String data) {
22         System.out.println("Transmitting data");
23     }
24
25     private void closeConnection() {
26         System.out.println("Closing connection");
27     }
28
29     // Public method - simple interface (ABSTRACTION!)
30     public void sendEmail(String from, String to, String message,
31         String password) {
32         System.out.println("==== Sending Email ====");
33         connectToServer("smtp.gmail.com");
34         authenticateUser(from, password);
35         prepareEmailHeaders(from, to);
36         encryptMessage(message);
37         transmitData(message);
38         closeConnection();
39         System.out.println("Email sent successfully!");
40     }
41
42 public class Main {
43     public static void main(String[] args) {
44         EmailService emailService = new EmailService();
```

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

← What user doesn't see

Student: How is this different from encapsulation again?

Expert: Let me show you side by side:

```
1 class BankAccount {  
2     // ENCAPSULATION – Hiding data  
3     private double balance; // Hidden data  
4  
5     public double getBalance() { // Controlled access  
6         return balance;  
7     }  
8  
9     public void setBalance(double balance) { // Controlled  
   modification  
10        if (balance >= 0) {  
11            this.balance = balance;  
12        }  
13    }  
14  
15    // ABSTRACTION – Hiding complexity  
16    private void validateAccount() {  
17        System.out.println("Validating account...");  
18    }  
19  
20    private void checkFraudDetection() {  
21        System.out.println("Checking for fraud...");  
22    }  
23  
24    private void updateTransactionLog() {  
25        System.out.println("Updating logs...");  
26    }  
27  
28    private void notifyUser() {  
29        System.out.println("Sending notification...");  
30    }  
31  
32    // Simple public interface (ABSTRACTION)  
33    public void withdraw(double amount) {  
34        System.out.println("==== Processing Withdrawal ====");  
35        validateAccount();  
36        checkFraudDetection();  
37  
38        if (amount > 0 && amount <= balance) {  
39            balance -= amount; // Using encapsulated data  
40            updateTransactionLog();  
41            notifyUser();  
42            System.out.println("Withdrawal successful: $" +
```

```

        amount);
43     } else {
44         System.out.println("Invalid amount or insufficient
balance!");
45     }
46 }
47 }

48 public class Main {
49     public static void main(String[] args) {
50         BankAccount account = new BankAccount();
51         account.setBalance(1000);
52
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     }
```

```
10
11     // Concrete method (has implementation)
12     public void sleep() {
13         System.out.println(name + " is sleeping...");
14     }
15
16     // Abstract method (no implementation) – MUST be implemented
17     // by subclass
17     public abstract void makeSound();
18
19     // Abstract method
20     public abstract void move();
21 }
23 // Concrete class – must implement all abstract methods
24 class Dog extends Animal {
25     public Dog(String name) {
26         super(name);
27     }
28
29     @Override
30     public void makeSound() {
31         System.out.println(name + " says: Woof! Woof!");
32     }
33
34     @Override
35     public void move() {
36         System.out.println(name + " runs on four legs");
37     }
38 }
39 class Bird extends Animal {
40     public Bird(String name) {
41         super(name);
42     }
43
44     @Override
45     public void makeSound() {
46         System.out.println(name + " says: Chirp! Chirp!");
47     }
48
49     @Override
50     public void move() {
51         System.out.println(name + " flies in the sky");
52     }
53 }
54 }
55 public class Main {
56     public static void main(String[] args) {
57         // Cannot do this:
58         // 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
23             failed!");
24         }
25     } else {
26         System.out.println("Cannot connect to payment
27         gateway!");
28     }
29
30     // Common implementation for all payment types
31     private void validatePayment() {
32         System.out.println("Validating payment amount: $" +
33         amount);
34         if (amount <= 0) {
35             throw new IllegalArgumentException("Invalid
36             amount!");
37         }
38
39     // Common implementation
40     private void sendConfirmation() {
41         System.out.println("Sending confirmation email...");
42     }
43
44     // Abstract methods – each payment type implements
45     // differently
46     protected abstract boolean connectToPaymentGateway();
47     protected abstract boolean authorizePayment();
48     protected abstract void capturePayment();
49 }
50
51 class CreditCardPayment extends PaymentProcessor {
52     private String cardNumber;
53
54     public CreditCardPayment(String merchantId, double amount,
55     String cardNumber) {
56         super(merchantId, amount);
57         this.cardNumber = cardNumber;
58     }
59
60     @Override
```

```
57     protected boolean connectToPaymentGateway() {
58         System.out.println("Connecting to Credit Card
59         Gateway...");
60         return true;
61     }
62
63     @Override
64     protected boolean authorizePayment() {
65         System.out.println("Authorizing credit card: " +
66         maskCardNumber());
67         return true;
68     }
69
70     @Override
71     protected void capturePayment() {
72         System.out.println("Capturing payment from credit card");
73     }
74
75     private String maskCardNumber() {
76         return "****-****-****-"
77         cardNumber.substring(cardNumber.length() - 4);
78     }
79
80 class PayPalPayment extends PaymentProcessor {
81     private String email;
82
83     public PayPalPayment(String merchantId, double amount, String
84     email) {
85         super(merchantId, amount);
86         this.email = email;
87     }
88
89     @Override
90     protected boolean connectToPaymentGateway() {
91         System.out.println("Connecting to PayPal Gateway...");
92         return true;
93     }
94
95     @Override
96     protected boolean authorizePayment() {
97         System.out.println("Authorizing PayPal account: " +
98         email);
99         return true;
100    }
101
102    @Override
103    protected void capturePayment() {
104        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,
107                           String walletAddress) {
108         super(merchantId, amount);
109         this.walletAddress = walletAddress;
110     }
111
112     @Override
113     protected boolean connectToPaymentGateway() {
114         System.out.println("Connecting to Bitcoin Network...");
115         return true;
116     }
117
118     @Override
119     protected boolean authorizePayment() {
120         System.out.println("Authorizing Bitcoin wallet: " +
121                           walletAddress);
122         return true;
123     }
124
125     @Override
126     protected void capturePayment() {
127         System.out.println("Capturing payment from Bitcoin
128                           wallet");
129     }
130
131     public class Main {
132         public static void main(String[] args) {
133             PaymentProcessor creditCard = new
134             CreditCardPayment("M123", 100.0, "1234567890123456");
135             creditCard.processPayment();
136
137             System.out.println("\n");
138
139             PaymentProcessor paypal = new PayPalPayment("M123",
140                 200.0, "user@example.com");
141             paypal.processPayment();
142
143             System.out.println("\n");
144
145             PaymentProcessor bitcoin = new BitcoinPayment("M123",
146                 300.0, "1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa");
147             bitcoin.processPayment();
148     }
149 }
```

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:

```
1 // Rule demonstrations
2 abstract class AbstractRules {
3
4     // ✅ Rule 1: Can have both abstract and concrete methods
5     public abstract void abstractMethod();
6
7     public void concreteMethod() {
8         System.out.println("This has implementation");
9     }
10
11    // ✅ Rule 2: Can have constructors
12    public AbstractRules() {
13        System.out.println("Abstract class constructor");
14    }
15
16    // ✅ Rule 3: Can have instance variables
17    private int value;
18
19    // ✅ Rule 4: Can have static methods
20    public static void staticMethod() {
21        System.out.println("Static method in abstract class");
22    }
23
24    // ✅ Rule 5: Can have final methods (cannot be overridden)
25    public final void finalMethod() {
26        System.out.println("Cannot override this");
27    }
28
29    // ✅ Rule 6: Abstract methods cannot be private
30    // private abstract void invalid(); // Compile error!
31
32    // ✅ Rule 7: Abstract methods cannot be final
33    // public final abstract void invalid(); // Compile error!
34
35    // ✅ Rule 8: Abstract methods cannot be static
36    // public static abstract void invalid(); // Compile error!
37 }
38 // ✅ Rule 9: Concrete subclass must implement ALL abstract
  methods
39 class ConcreteClass extends AbstractRules {
40     @Override
41     public void abstractMethod() {
42         System.out.println("Implemented!");
43     }
44 }
45
46 // ✅ Rule 10: Abstract subclass doesn't need to implement
  abstract methods
```

```

48 abstract class AnotherAbstractClass extends AbstractRules {
49     // Can add more abstract methods
50     public abstract void anotherMethod();
51 }
52 public class Main {
53     public static void main(String[] args) {
54         // ❌ Rule 11: Cannot instantiate abstract class
55         // AbstractRules obj = new AbstractRules(); // Compile
56         // error!
57
58         // ✅ But can create reference of abstract type
59         AbstractRules obj = new ConcreteClass();
60         obj.abstractMethod();
61         obj.concreteMethod();
62         obj.finalMethod();
63
64         // ✅ Can call static methods using class name
65         AbstractRules.staticMethod();
66     }
67 }

```

Abstraction Levels

Student: Can you show me different levels of abstraction?

Expert: Great question! Let me demonstrate:

```

1 // High-level abstraction
2 abstract class Vehicle {
3     public abstract void start();
4     public abstract void stop();
5 }
6 // Medium-level abstraction
7 abstract class Car extends Vehicle {
8     protected int numberOfWorkers;
9
10    public abstract void openTrunk();
11
12    // Partially implemented
13    @Override
14    public void start() {
15        System.out.println("Turning key...");
16        startEngine(); // Abstract - subclass decides
17    }
18
19

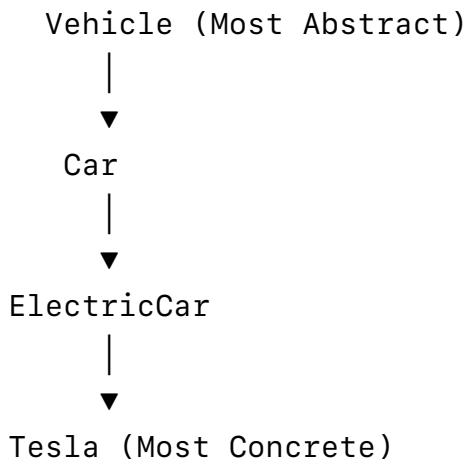
```

```

20     protected abstract void startEngine();
21 }
23 // Low-level abstraction (more specific)
24 abstract class ElectricCar extends Car {
25     protected int batteryCapacity;
26
27     @Override
28     protected void startEngine() {
29         System.out.println("Powering electric motor...");
30     }
31
32     public abstract void chargeBattery();
33 }
35 // Concrete implementation (no abstraction)
36 class Tesla extends ElectricCar {
37     public Tesla() {
38         this.numberOfDoors = 4;
39         this.batteryCapacity = 100;
40     }
41
42     @Override
43     public void stop() {
44         System.out.println("Regenerative braking engaged");
45     }
46
47     @Override
48     public void openTrunk() {
49         System.out.println("Trunk opening automatically");
50     }
51
52     @Override
53     public void chargeBattery() {
54         System.out.println("Supercharging at 250kW");
55     }
56 }
58 public class Main {
59     public static void main(String[] args) {
60         Tesla myTesla = new Tesla();
61
62         myTesla.start();           // From Car
63         myTesla.openTrunk();       // From Tesla
64         myTesla.chargeBattery();  // From Tesla
65         myTesla.stop();          // From Tesla
66     }
67 }

```

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."

```

1 abstract class Employee {
2     protected String name;
3     protected String id;
4
5     // Constructor in abstract class
6     public Employee(String name, String id) {
7         this.name = name;
8         this.id = id;
9         System.out.println("Employee constructor called");
10    }
11
12    public abstract double calculateSalary();
13 }
15 class FullTimeEmployee extends Employee {
16     private double monthlySalary;
17
18     public FullTimeEmployee(String name, String id, double
monthlySalary) {
19         super(name, id); // Calls abstract class constructor
20         this.monthlySalary = monthlySalary;
21         System.out.println("FullTimeEmployee constructor
called");
22    }
23
24    @Override
25    public double calculateSalary() {
26        return monthlySalary;
27    }
28 }
30 public class Main {
31     public static void main(String[] args) {
32         Employee emp = new FullTimeEmployee("John", "E001",
5000);
33         // Output:
34         // Employee constructor called
35         // FullTimeEmployee constructor called
36    }
37 }

```

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.

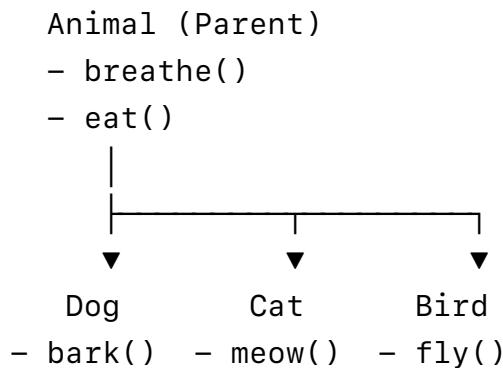
Chapter 5: Inheritance

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:



Student: Can you show me a code example?

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

```
1 // Parent class (also called superclass or base class)
2 class Animal {
3     String name;
4     int age;
5
6     public void eat() {
7         System.out.println(name + " is eating...");
8     }
9
10    public void sleep() {
11        System.out.println(name + " is sleeping...");
12    }
13 }
15 // Child class (also called subclass or derived class)
16 class Dog extends Animal {
17     String breed;
18
19     public void bark() {
20         System.out.println(name + " says: Woof! Woof!");
21     }
22
23     public void wagTail() {
24         System.out.println(name + " is wagging tail!");
25     }
26 }
28 class Cat extends Animal {
29     String color;
```

```
31     public void meow() {
32         System.out.println(name + " says: Meow! Meow!");
33     }
34
35     public void scratch() {
36         System.out.println(name + " is scratching!");
37     }
38 }
39
40 public class Main {
41     public static void main(String[] args) {
42         Dog dog = new Dog();
43         dog.name = "Buddy";           // Inherited from Animal
44         dog.age = 3;                 // Inherited from Animal
45         dog.breed = "Labrador";      // Dog's own property
46
47         dog.eat();                  // Inherited method
48         dog.sleep();                // Inherited method
49         dog.bark();                 // Dog's own method
50         dog.wagTail();              // Dog's own method
51
52         System.out.println();
53
54         Cat cat = new Cat();
55         cat.name = "Whiskers";      // Inherited from Animal
56         cat.age = 2;                 // Inherited from Animal
57         cat.color = "Orange";       // Cat's own property
58
59         cat.eat();                  // Inherited method
60         cat.sleep();                // Inherited method
61         cat.meow();                 // Cat's own method
62         cat.scratch();              // Cat's own method
63     }
64 }
```

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:

```

1 class Parent {
2     // ✅ Inherited by child
3     public int publicField = 1;
4     protected int protectedField = 2;
5
6     // ❌ NOT inherited by child
7     private int privateField = 3;
8
9     // ✅ Inherited
10    public void publicMethod() {
11        System.out.println("Public method");
12    }
13
14    // ✅ Inherited
15    protected void protectedMethod() {
16        System.out.println("Protected method");
17    }
18
19    // ❌ NOT inherited
20    private void privateMethod() {
21        System.out.println("Private method");
22    }
23 }
24
25 class Child extends Parent {
26     public void testAccess() {
27         System.out.println(publicField);      // ✅ OK
28         System.out.println(protectedField);  // ✅ OK
29         // System.out.println(privateField); // ❌ Compile
30             error!
31
32         publicMethod();      // ✅ OK
33         protectedMethod();  // ✅ OK
34         // privateMethod(); // ❌ Compile error!
35     }
36 }
37
38 public class Main {
39     public static void main(String[] args) {
40         Child child = new Child();
41         child.testAccess();
42     }
43 }
```

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

```

1 // One parent, one child
2 class Vehicle {
3     void start() {
4         System.out.println("Vehicle starting...");}
5     }
6 }
7 class Car extends Vehicle {
8     void drive() {
9         System.out.println("Car driving...");}
10    }
11 }
12 }
13 public class Main {
14     public static void main(String[] args) {
15         Car car = new Car();
16         car.start(); // From Vehicle
17         car.drive(); // From Car
18     }
19 }
20 }
```

Diagram:

```
Vehicle
```

```
|
```

```
▼
```

```
Car
```

2. Multilevel Inheritance

```
1 // Chain of inheritance
2 class Vehicle {
3     void start() {
4         System.out.println("Vehicle starting...");
```

```
5     }
6 }
7 class Car extends Vehicle {
8     void drive() {
9         System.out.println("Car driving...");
```

```
10    }
11 }
12 }
13 class SportsCar extends Car {
14     void turboBoost() {
15         System.out.println("Turbo boost activated!");
```

```
16    }
17 }
18 }
19 public class Main {
20     public static void main(String[] args) {
21         SportsCar sportsCar = new SportsCar();
22         sportsCar.start();           // From Vehicle
23         sportsCar.drive();         // From Car
24         sportsCar.turboBoost();    // From SportsCar
25     }
26 }
27 }
```

Diagram:

```
Vehicle
```

```
|
```

```
▼
```

```
Car
```

```
|
```

```
▼
```

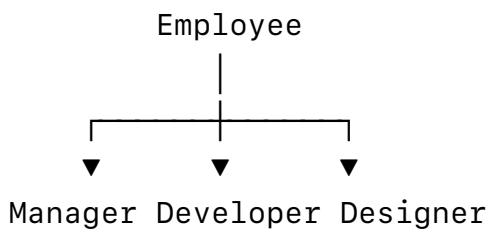
```
SportsCar
```

3. Hierarchical Inheritance

```
1 // Multiple children from one parent
2 class Employee {
3     String name;
4     double baseSalary;
5
6     void work() {
7         System.out.println(name + " is working...");
8     }
9 }
10 class Manager extends Employee {
11     int teamSize;
12
13     void conductMeeting() {
14         System.out.println(name + " is conducting a meeting");
15     }
16 }
17 }
18 class Developer extends Employee {
19     String programmingLanguage;
20
21     void writeCode() {
22         System.out.println(name + " is writing " +
23             programmingLanguage + " code");
24     }
25 }
26 class Designer extends Employee {
27     String designTool;
28
29     void createDesign() {
30         System.out.println(name + " is designing with " +
31             designTool);
32     }
33 }
34 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!)

```

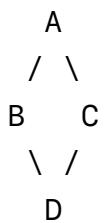
1 // X This is NOT allowed in Java!
2 /*
3 class Father {
4     void property() {
5         System.out.println("Father's property");
6     }
7 }
8 class Mother {
9     void property() {
10        System.out.println("Mother's property");
11    }
12 }
13 */
14 // This causes "Diamond Problem"
15 class Child extends Father, Mother { // X Compile error!
16     // Which property() method should Child inherit?
17 }
18 */
19 */

```

Student: Why doesn't Java support multiple inheritance?

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

Diamond Problem:



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:

```
1 class Parent {
2     String parentName;
3
4     // Parent constructor
5     public Parent() {
6         System.out.println("Parent constructor called");
7         parentName = "Parent";
8     }
9
10    public Parent(String name) {
11        System.out.println("Parent parameterized constructor
12 called");
13        this.parentName = name;
14    }
15 class Child extends Parent {
16     String childName;
17
18     // Child constructor
19     public Child() {
20         // super() is automatically called here (invisible)
21         System.out.println("Child constructor called");
22         childName = "Child";
23     }
24
25
26     public Child(String parentName, String childName) {
27         super(parentName); // Explicitly calling parent
28 constructor
29         System.out.println("Child parameterized constructor
30 called");
31         this.childName = childName;
32     }
33 }
34
35 public class Main
36 {
37     public static void main(String[] args) {
38         System.out.println("== Creating Child with default
39 constructor ==");
40         Child child1 = new Child();
41
42         System.out.println("\n== Creating Child with
43 parameterized constructor ==");
44         Child child2 = new Child("SuperParent", "SuperChild");
45
46         System.out.println("\nParent name: " +
47 child2.parentName);
48         System.out.println("Child name: " + child2.childName);
49 }
```

```
44      }
45 }
```

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:

```
1 class Parent {  
2     public Parent() {  
3         System.out.println("Parent no-arg constructor");  
4     }  
5  
6     public Parent(String name) {  
7         System.out.println("Parent parameterized constructor: " +  
8             name);  
9     }  
10    class Child extends Parent {  
12        // This constructor:  
13        public Child() {  
14            System.out.println("Child constructor");  
15        }  
16  
17        // Is automatically converted to:  
18        /*  
19        public Child() {  
20            super(); // Automatically added by Java  
21            System.out.println("Child constructor");  
22        }  
23        */  
24    }  
25    // ✘ This will cause an error:  
26    class ProblematicChild extends Parent {  
27        public ProblematicChild(String name) {  
28            // Java tries to call super() automatically  
29            // But Parent doesn't have a no-arg constructor anymore!  
30            System.out.println("Child: " + name);  
31        }  
32    }  
33 }
```

To fix the error:

```
1 class FixedChild extends Parent {  
2     public FixedChild(String name) {  
3         super(name); // Explicitly call parameterized parent  
4             constructor  
5             System.out.println("Child: " + name);  
6     }  
7 }
```

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:

```
1 class Animal {  
2     void makeSound() {  
3         System.out.println("Animal makes a sound");  
4     }  
5  
6     void eat() {  
7         System.out.println("Animal is eating");  
8     }  
9 }  
10 class Dog extends Animal {  
11     // Overriding the makeSound method  
12     @Override  
13     void makeSound() {  
14         System.out.println("Dog says: Woof! Woof!");  
15     }  
16  
17     // Inherited eat() method is not overridden  
18 }  
19  
20 class Cat extends Animal {  
21     // Overriding the makeSound method  
22     @Override  
23     void makeSound() {  
24         System.out.println("Cat says: Meow! Meow!");  
25     }  
26  
27     // Overriding the eat method too  
28     @Override  
29     void eat() {  
30         System.out.println("Cat is eating fish");  
31     }  
32 }  
33  
34 public class Main {  
35     public static void main(String[] args) {  
36         Animal animal = new Animal();  
37         animal.makeSound(); // Animal makes a sound  
38         animal.eat(); // Animal is eating  
39  
40         System.out.println();  
41  
42         Dog dog = new Dog();
```

```

44     dog.makeSound();    // Dog says: Woof! Woof! (Overridden)
45     dog.eat();          // Animal is eating (Inherited)
46
47     System.out.println();
48
49     Cat cat = new Cat();
50     cat.makeSound();   // Cat says: Meow! Meow! (Overridden)
51     cat.eat();         // Cat is eating fish (Overridden)
52 }
53 }
```

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:

```

1 class Parent {
2     void display() {
3         System.out.println("Parent display");
4     }
5 }
6 class Child extends Parent {
7     // ✅ Correct override
8     @Override
9     void display() {
10         System.out.println("Child display");
11     }
12
13
14     // ❌ Typo – compiler will catch this because of @Override
15     /*
16     @Override
17     void displaya() { // Compile error: method doesn't override
18         System.out.println("Child display");
19     }
20     */
21 }
```

Rules for Method Overriding

Expert: Here are the important rules:

```
1 class Parent {
```

```
2 // Original method
3 protected String getMessage(int value) {
4     return "Parent: " + value;
5 }
6
7 public void display() {
8     System.out.println("Parent display");
9 }
10
11 public final void cannotOverride() {
12     System.out.println("This cannot be overridden");
13 }
14 }
15 class Child extends Parent {
16     //  Rule 1: Same method signature (name + parameters)
17     @Override
18     protected String getMessage(int value) {
19         return "Child: " + value;
20     }
21
22     //  Rule 2: Return type must be same or covariant
23     // (subtype)
24     @Override
25     public void display() {
26         System.out.println("Child display");
27     }
28
29     //  Rule 3: Access modifier can be same or more accessible
30     // protected -> public is OK
31     // public -> protected is NOT OK
32
33     //  Rule 4: Cannot override final methods
34     /*
35     @Override
36     public void cannotOverride() { // Compile error!
37         System.out.println("Trying to override");
38     }
39     */
40
41     //  Rule 5: Cannot reduce access level
42     /*
43     @Override
44     private String getMessage(int value) { // Compile error!
45         return "Child: " + value;
46     }
47     */
48 }
```

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:

```
1 // Base class
2 class Employee {
3     protected String name;
4     protected String employeeId;
5     protected double baseSalary;
6
7     public Employee(String name, String employeeId, double
baseSalary) {
8         this.name = name;
9         this.employeeId = employeeId;
10        this.baseSalary = baseSalary;
11    }
12
13    // Method to be overridden
14    public double calculateSalary() {
15        return baseSalary;
16    }
17
18    // Method to be overridden
19    public void displayInfo() {
20        System.out.println("Name: " + name);
21        System.out.println("ID: " + employeeId);
22        System.out.println("Salary: $" + calculateSalary());
23    }
```

```
24
25     // Common method (not overridden)
26     public void clockIn() {
27         System.out.println(name + " clocked in at " +
28             java.time.LocalTime.now());
29     }
30 // Full-time employee
32 class FullTimeEmployee extends Employee {
33     private double bonus;
34     private double benefits;
35
36     public FullTimeEmployee(String name, String employeeId,
37         double baseSalary, double bonus, double benefits) {
37         super(name, employeeId, baseSalary);
38         this.bonus = bonus;
39         this.benefits = benefits;
40     }
41
42     @Override
43     public double calculateSalary() {
44         return baseSalary + bonus + benefits;
45     }
46
47     @Override
48     public void displayInfo() {
49         System.out.println("== Full-Time Employee ==");
50         super.displayInfo(); // Call parent's displayInfo
51         System.out.println("Bonus: $" + bonus);
52         System.out.println("Benefits: $" + benefits);
53     }
54 }
55 // Part-time employee
57 class PartTimeEmployee extends Employee {
58     private int hoursWorked;
59     private double hourlyRate;
60
61     public PartTimeEmployee(String name, String employeeId, int
62         hoursWorked, double hourlyRate) {
62         super(name, employeeId, 0); // No base salary
63         this.hoursWorked = hoursWorked;
64         this.hourlyRate = hourlyRate;
65     }
66
67     @Override
68     public double calculateSalary() {
69         return hoursWorked * hourlyRate;
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: $" +
80             calculateSalary());
81     }
82 // Contractor
83 class Contractor extends Employee {
84     private int projectsCompleted;
85     private double ratePerProject;
86
87     public Contractor(String name, String employeeId, int
88         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: " +
105            projectsCompleted);
106        System.out.println("Rate per Project: $" +
107            ratePerProject);
108        System.out.println("Total Payment: $" +
109            calculateSalary());
110    }
111 }
112 public static void main(String[] args) {
113     // Create different types of employees
114     FullTimeEmployee fte = new FullTimeEmployee("Alice",
115         "FT001", 5000, 1000, 500);
116     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:

```
1 class Parent {  
2     String name = "Parent";  
3 }
```

```

4  public Parent() {
5      System.out.println("Parent constructor");
6  }
7
8  public Parent(String name) {
9      this.name = name;
10     System.out.println("Parent constructor with name: " +
11         name);
12 }
13
14 public void display() {
15     System.out.println("Parent display");
16 }
17
18 class Child extends Parent {
19     String name = "Child";
20
21     public Child() {
22         // Use 1: Call parent constructor
23         super("Custom Parent"); // Must be first line
24         System.out.println("Child constructor");
25     }
26
27     public void display() {
28         // Use 2: Call parent method
29         super.display();
30         System.out.println("Child display");
31     }
32
33     public void showNames() {
34         // Use 3: Access parent variable
35         System.out.println("Child name: " + this.name);
36         System.out.println("Parent name: " + super.name);
37     }
38 }
39
40 public class Main {
41     public static void main(String[] args) {
42         Child child = new Child();
43         System.out.println();
44
45         child.display();
46         System.out.println();
47
48         child.showNames();
49     }
50 }

```

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:

```

1 class Animal {
2     void breathe() {
3         System.out.println("Breathing...");
4     }
5 }
6 class Dog extends Animal {
7     void bark() {
8         System.out.println("Woof!");
9     }
10 }
11 }
12 public class Main {
13     public static void main(String[] args) {
14         Dog dog = new Dog();
15
16         // Dog IS-A Animal (inheritance relationship)
17         // So we can do this:
18         Animal animal = dog; // Upcasting
19         animal.breathe(); // Works fine
20         // animal.bark(); // Compile error - Animal doesn't
21         have bark()
22
23         // We can check the relationship:
24         System.out.println("dog instanceof Dog: " + (dog
25             instanceof Dog)); // true
26         System.out.println("dog instanceof Animal: " + (dog
27             instanceof Animal)); // true
28         System.out.println("animal instanceof Dog: " + (animal
29             instanceof Dog)); // true
30         System.out.println("animal instanceof Animal: " + (animal
31             instanceof Animal)); // true
32     }
33 }

```

IS-A Test:

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

- | | |
|---|-----------------------------|
| <input checked="" type="checkbox"/> Dog IS-A Animal | → Dog extends Animal |
| <input checked="" type="checkbox"/> Car IS-A Vehicle | → Car extends Vehicle |
| <input checked="" type="checkbox"/> Manager IS-A Employee | → Manager extends Employee |
| <input checked="" type="checkbox"/> Car IS-A Wheel | → Wrong! (Car HAS-A Wheel) |
| <input checked="" type="checkbox"/> 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:

```
1 //  GOOD use of inheritance - Clear IS-A relationship
2 class Shape {
3     protected String color;
4
5     public double calculateArea() {
6         return 0;
7     }
8 }
9
10 class Circle extends Shape {
11     private double radius;
12
13     @Override
14     public double calculateArea() {
15         return Math.PI * radius * radius;
16     }
17 }
18 class Rectangle extends Shape {
19     private double length;
20     private double width;
21
22     @Override
23     public double calculateArea() {
24         return length * width;
25     }
26 }
27 }
28 // Circle IS-A Shape 
29 // Rectangle IS-A Shape 
```

```

1 // ❌ BAD use of inheritance - No IS-A relationship
2 class ArrayList {
3     private Object[] elements;
4
5     public void add(Object obj) {
6         // Add logic
7     }
8 }
10 // Stack IS-A ArrayList? No! This is wrong!
11 class Stack extends ArrayList {
12     public void push(Object obj) {
13         add(obj);
14     }
15 }
16 // 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."

```
1 class Parent {  
2     private void display() {  
3         System.out.println("Parent private display");  
4     }  
5  
6     public void callDisplay() {  
7         display(); // Calls Parent's private display  
8     }  
9 }  
10 class Child extends Parent {  
11     // This is NOT overriding - it's a new method  
12     private void display() {  
13         System.out.println("Child private display");  
14     }  
15 }  
16  
17     public void callChildDisplay() {  
18         display(); // Calls Child's own display  
19     }  
20 }  
21 public class Main {  
22     public static void main(String[] args) {  
23         Child child = new Child();  
24         child.callDisplay(); // Parent private display  
25         child.callChildDisplay(); // Child private display  
26     }  
27 }  
28 }
```

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."

```

1 class Parent {
2     public static void staticMethod() {
3         System.out.println("Parent static method");
4     }
5 }
6 class Child extends Parent {
7     // This is method HIDING, not overriding
8     public static void staticMethod() {
9         System.out.println("Child static method");
10    }
11 }
12 }
13 public class Main {
14     public static void main(String[] args) {
15         Parent.staticMethod(); // Parent static method
16         Child.staticMethod(); // Child static method
17
18         Parent obj = new Child();
19         obj.staticMethod(); // Parent static method (NOT
polymorphic!)
20     }
21 }
22 }
```

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:

```

1 class Animal {
2     void makeSound() {
3         System.out.println("Animal makes a sound");
4     }
5 }
6 class Dog extends Animal {
7     @Override
8     void makeSound() {
9         System.out.println("Woof! Woof!");
10    }
11 }
12 }
13 class Cat extends Animal {
14     @Override
15     void makeSound() {
16         System.out.println("Meow! Meow!");
17     }
18 }
19 }
20 class Cow extends Animal {
21     @Override
22     void makeSound() {
23         System.out.println("Moo! Moo!");
24     }
25 }
26 }
27 public class Main {
28     public static void main(String[] args) {
29         // Same reference type (Animal), different objects
30         Animal animal1 = new Dog();
31         Animal animal2 = new Cat();
32         Animal animal3 = new Cow();
33
34         // Same method call, different behavior – POLYMORPHISM!
35         animal1.makeSound(); // Woof! Woof!
36         animal2.makeSound(); // Meow! Meow!
37         animal3.makeSound(); // Moo! Moo!
38     }
39 }
40 }

```

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:

```
1 class Animal {
2     String name;
3
4     Animal(String name) {
5         this.name = name;
6     }
7
8     void makeSound() {
9         System.out.println(name + " makes a sound");
10    }
11
12    void eat() {
13        System.out.println(name + " is eating");
14    }
15 }
16 class Dog extends Animal {
17     Dog(String name) {
18         super(name);
19     }
20
21     @Override
22     void makeSound() {
23         System.out.println(name + " says: Woof!");
24     }
25 }
26
27 class Cat extends Animal {
28     Cat(String name) {
29         super(name);
30     }
31
32     @Override
33     void makeSound() {
34         System.out.println(name + " says: Meow!");
35     }
36 }
37 }
38 public class Main {
39     // One method works with all animal types!
40     public static void makeAnimalSpeak(Animal animal) {
41         animal.makeSound(); // Polymorphic call
42     }
43 }
44
45 public static void feedAnimal(Animal animal) {
46     animal.eat();
```

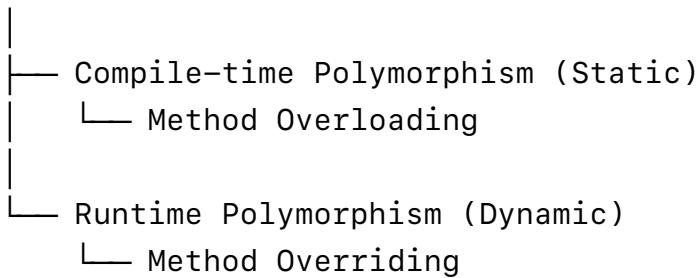
```
47     }
48
49     public static void main(String[] args) {
50         Dog dog = new Dog("Buddy");
51         Cat cat = new Cat("Whiskers");
52
53         // Same method, different animals
54         makeAnimalSpeak(dog); // Buddy says: Woof!
55         makeAnimalSpeak(cat); // Whiskers says: Meow!
56
57         feedAnimal(dog); // Buddy is eating
58         feedAnimal(cat); // Whiskers is eating
59     }
60 }
```

Types of Polymorphism

Student: Are there different types of polymorphism?

Expert: Yes! There are two main types:

Polymorphism



1. Compile-time Polymorphism (Method Overloading)

```

1 class Calculator {
2     // Same method name, different parameters
3
4     // Method 1: Two integers
5     int add(int a, int b) {
6         return a + b;
7     }
8
9     // Method 2: Three integers
10    int add(int a, int b, int c) {
11        return a + b + c;
12    }
13
14    // Method 3: Two doubles
15    double add(double a, double b) {
16        return a + b;
17    }
18
19    // Method 4: String concatenation
20    String add(String a, String b) {
21        return a + b;
22    }
23 }
24 public class Main {
25     public static void main(String[] args) {
26         Calculator calc = new Calculator();
27
28         System.out.println(calc.add(5, 10)); // 15
29         (Method 1)
30         System.out.println(calc.add(5, 10, 15)); // 30
31         (Method 2)
32         System.out.println(calc.add(5.5, 10.5)); // 16.0
33         (Method 3)
34         System.out.println(calc.add("Hello", "World")); // HelloWorld
35         (Method 4)
36     }
37 }

```

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

```

1 class Demo {
2     void display(int a) {
3         System.out.println("Integer: " + a);
4     }
5
6     void display(String a) {
7         System.out.println("String: " + a);
8     }
9
10    void display(int a, String b) {
11        System.out.println("Int and String: " + a + ", " + b);
12    }
13
14    void display(String a, int b) {
15        System.out.println("String and Int: " + a + ", " + b);
16    }
17 }
18 public class Main {
19     public static void main(String[] args) {
20         Demo demo = new Demo();
21
22         demo.display(10);           // Calls display(int)
23         demo.display("Hello");    // Calls display(String)
24         demo.display(10, "World"); // Calls display(int,
25                               String)
26         demo.display("Hello", 20); // Calls display(String,
27                               int)
28     }

```

2. Runtime Polymorphism (Method Overriding)

```

1 class Shape {
2     void draw() {
3         System.out.println("Drawing a shape");
4     }
5
6     double calculateArea() {
7         return 0;
8     }
9 }
10 class Circle extends Shape {
11     double radius;
12
13

```

```
14     Circle(double radius) {
15         this.radius = radius;
16     }
17
18     @Override
19     void draw() {
20         System.out.println("Drawing a circle");
21     }
22
23     @Override
24     double calculateArea() {
25         return Math.PI * radius * radius;
26     }
27 }
28 class Rectangle extends Shape {
29     double length, width;
30
31     Rectangle(double length, double width) {
32         this.length = length;
33         this.width = width;
34     }
35
36     @Override
37     void draw() {
38         System.out.println("Drawing a rectangle");
39     }
40
41     @Override
42     double calculateArea() {
43         return length * width;
44     }
45 }
46 }
47 public class Main {
48     public static void main(String[] args) {
49         // Reference type: Shape
50         // Object type: Circle, Rectangle (decided at runtime)
51         Shape shape1 = new Circle(5);
52         Shape shape2 = new Rectangle(4, 6);
53
54         // Method call resolved at RUNTIME
55         shape1.draw(); // Drawing a circle
56         System.out.println("Area: " + shape1.calculateArea());
57         // 78.53...
58
59         shape2.draw(); // Drawing a rectangle
60         System.out.println("Area: " + shape2.calculateArea());
61         // 24.0
62     }
63 }
```

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

Expert: Let me show you:

Compile-time Polymorphism

- Method Overloading
- Same class
- Same method name
- Different parameters
- Resolved at compile-time
- Static binding
- Faster

Runtime Polymorphism

- Method Overriding
- Different classes
- Same method name
- Same parameters
- Resolved at runtime
- Dynamic binding
- 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:

```

1 class Animal {
2     void eat() {
3         System.out.println("Animal is eating");
4     }
5 }
6 class Dog extends Animal {
7     @Override
8     void eat() {
9         System.out.println("Dog is eating");
10    }
11 }
12
13     void bark() {
14         System.out.println("Dog is barking");
15     }
16 }
17 public class Main {
18     public static void main(String[] args) {

```

```

20      // UPCASTING - Implicit (automatic)
21      // Child to Parent reference
22      Animal animal = new Dog(); // Upcasting happens
23      automatically
24      animal.eat(); // Works - method is overridden
25      // animal.bark(); // ✗ Compile error - Animal doesn't
26      have bark()
27      // DOWNCASTING - Explicit (manual)
28      // Parent to Child reference
29      Dog dog = (Dog) animal; // Need explicit cast
30      dog.eat(); // Works
31      dog.bark(); // Works now!
32
33      System.out.println();
34
35      // ! DANGEROUS DOWNCASTING
36      Animal animal2 = new Animal(); // Actually an Animal
37      object
38      // Dog dog2 = (Dog) animal2; // ✗ ClassCastException
39      at runtime!
40
41      // SAFE DOWNCASTING - Check before casting
42      if (animal2 instanceof Dog) {
43          Dog dog2 = (Dog) animal2;
44          dog2.bark();
45      } else {
46          System.out.println("animal2 is not a Dog!");
47      }

```

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:

```
1 class Animal {  
2     void makeSound() {  
3         System.out.println("Some sound");  
4     }  
5 }  
6 class Dog extends Animal {  
7     @Override  
8     void makeSound() {  
9         System.out.println("Woof!");  
10    }  
11 }  
12  
13     void fetch() {  
14         System.out.println("Fetching ball");  
15     }  
16 }  
17 class Cat extends Animal {  
18     @Override  
19     void makeSound() {  
20         System.out.println("Meow!");  
21     }  
22 }  
23  
24     void scratch() {  
25         System.out.println("Scratching");  
26     }  
27 }  
28 public class Main {  
29     public static void handleAnimal(Animal animal) {  
30         // Safe – available in Animal class  
31         animal.makeSound();  
32     }  
33 }
```

```
33
34     // Check type before calling specific methods
35     if (animal instanceof Dog) {
36         Dog dog = (Dog) animal;
37         dog.fetch();
38     } else if (animal instanceof Cat) {
39         Cat cat = (Cat) animal;
40         cat.scratch();
41     }
42 }
43
44 public static void main(String[] args) {
45     Animal dog = new Dog();
46     Animal cat = new Cat();
47
48     handleAnimal(dog);
49     System.out.println();
50     handleAnimal(cat);
51 }
52 }
```

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:

```
1 // Base class
2 abstract class Payment {
3     protected String transactionId;
4     protected double amount;
5     protected String customerName;
6
7     public Payment(String transactionId, double amount, String
```

```
    customerName) {
8        this.transactionId = transactionId;
9        this.amount = amount;
10       this.customerName = customerName;
11    }
12
13 // Template method – same for all payments
14 public final void processPayment() {
15     System.out.println("==== Processing Payment ====");
16     System.out.println("Transaction ID: " + transactionId);
17     System.out.println("Customer: " + customerName);
18     System.out.println("Amount: $" + amount);
19
20     if (validatePayment()) {
21         if (executePayment()) {
22             generateReceipt();
23             sendConfirmation();
24             System.out.println("Payment successful!");
25         } else {
26             System.out.println("Payment failed!");
27         }
28     } else {
29         System.out.println("Payment validation failed!");
30     }
31     System.out.println();
32 }
33
34 // Abstract methods – different for each payment type
35 protected abstract boolean validatePayment();
36 protected abstract boolean executePayment();
37
38 // Common methods
39 private void generateReceipt() {
40     System.out.println("Receipt generated");
41 }
42
43 private void sendConfirmation() {
44     System.out.println("Confirmation email sent");
45 }
46 }
47 // Credit Card Payment
48 class CreditCardPayment extends Payment {
49     private String cardNumber;
50     private String cvv;
51     private String expiryDate;
52
53     public CreditCardPayment(String transactionId, double amount,
54     String customerName,
```

```
55                               String cardNumber, String cvv, String
56           expiryDate) {
57               super(transactionId, amount, customerName);
58               this.cardNumber = cardNumber;
59               this.cvv = cvv;
60               this.expiryDate = expiryDate;
61           }
62
63   @Override
64   protected boolean validatePayment() {
65       System.out.println("Validating credit card...");
66       // Simplified validation
67       return cardNumber.length() == 16 && cvv.length() == 3;
68   }
69
70   @Override
71   protected boolean executePayment() {
72       System.out.println("Charging credit card: *****-*****-"
73           +
74           cardNumber.substring(12));
75   }
76 // PayPal Payment
77 class PayPalPayment extends Payment {
78     private String email;
79     private String password;
80
81     public PayPalPayment(String transactionId, double amount,
82         String customerName,
83             String email, String password) {
84         super(transactionId, amount, customerName);
85         this.email = email;
86         this.password = password;
87     }
88
89     @Override
90     protected boolean validatePayment() {
91         System.out.println("Validating PayPal account...");
92         return email.contains("@") && password.length() >= 6;
93     }
94
95     @Override
96     protected boolean executePayment() {
97         System.out.println("Processing PayPal payment for: " +
98             email);
99         return true;
100    }
```

```
100 }
101 // Bank Transfer Payment
102 class BankTransferPayment extends Payment {
103     private String accountNumber;
104     private String ifscCode;
105
106     public BankTransferPayment(String transactionId, double
107         amount, String customerName,
108             String accountNumber, String
109             ifscCode) {
110         super(transactionId, amount, customerName);
111         this.accountNumber = accountNumber;
112         this.ifscCode = ifscCode;
113     }
114
115     @Override
116     protected boolean validatePayment() {
117         System.out.println("Validating bank account...");
```

118 return accountNumber.length() >= 10 && ifscCode.length()

119 == 11;

120 }

121
122 @Override
123 protected boolean executePayment() {
124 System.out.println("Initiating bank transfer from
125 account: " +
126 "****" +
127 accountNumber.substring(accountNumber.length() - 4));
128 return true;
129 }
130 }
131
132 // Cryptocurrency Payment
133 class CryptoPayment extends Payment {
134 private String walletAddress;
135 private String cryptoType;
136
137 public CryptoPayment(String transactionId, double amount,
138 String customerName,
139 String walletAddress, String cryptoType)
140 {
141 super(transactionId, amount, customerName);
142 this.walletAddress = walletAddress;
143 this.cryptoType = cryptoType;
144 }
145
146 @Override
147 protected boolean validatePayment() {
148 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 // Payment Processor  
154 class PaymentProcessor {  
155     // Polymorphic method - accepts any Payment type  
156     public void process(Payment payment) {  
157         payment.processPayment();  
158     }  
159  
160     // Process multiple payments  
161     public void processBatch(Payment[] payments) {  
162         System.out.println("==> Batch Processing " +  
payments.length + " Payments ==\n");  
163         double totalAmount = 0;  
164  
165         for (Payment payment : payments) {  
166             payment.processPayment();  
167             totalAmount += payment.amount;  
168         }  
169     }  
170  
171     System.out.println("Total amount processed: $" +  
totalAmount);  
172     }  
173 }  
174 public class Main {  
175     public static void main(String[] args) {  
176         // Create different payment types  
177         Payment creditCard = new CreditCardPayment(  
178             "TXN001", 150.00, "John Doe",  
179             "1234567890123456", "123", "12/25"  
180         );  
181  
182         Payment paypal = new PayPalPayment(  
183             "TXN002", 200.00, "Jane Smith",  
184             "jane@example.com", "password123"  
185         );  
186     }  
187 }
```

```

188     Payment bankTransfer = new BankTransferPayment(
189         "TXN003", 500.00, "Bob Johnson",
190         "1234567890", "ABCD0123456"
191     );
192
193     Payment crypto = new CryptoPayment(
194         "TXN004", 1000.00, "Alice Williams",
195         "1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa", "Bitcoin"
196     );
197
198     // Process individual payments
199     PaymentProcessor processor = new PaymentProcessor();
200
201     processor.process(creditCard);
202     processor.process(paypal);
203     processor.process(bankTransfer);
204     processor.process(crypto);
205
206     // Batch processing
207     Payment[] payments = {creditCard, paypal, bankTransfer,
208     crypto};
209     processor.processBatch(payments);
210 }
```

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:

```
1 class Vehicle {  
2     void start() {  
3         System.out.println("Vehicle is starting");  
4     }  
5  
6     void stop() {
```

```

7           System.out.println("Vehicle is stopping");
8     }
9 }
10 class Car extends Vehicle {
11     @Override
12     void start() {
13         System.out.println("Car is starting with key");
14     }
15
16
17 // stop() is inherited, not overridden
18 }
19 class Bike extends Vehicle {
20     @Override
21     void start() {
22         System.out.println("Bike is starting with kick");
23     }
24
25
26     @Override
27     void stop() {
28         System.out.println("Bike is stopping with hand brake");
29     }
30 }
31 public class Main {
32     public static void main(String[] args) {
33         // Reference type: Vehicle
34         // Object type: determined at runtime
35
36         Vehicle v1 = new Car();
37         Vehicle v2 = new Bike();
38         Vehicle v3 = new Vehicle();
39
40         System.out.println("== v1 (Car object) ==");
41         v1.start(); // Car's overridden method
42         v1.stop(); // Vehicle's method (not overridden in Car)
43
44         System.out.println("\n== v2 (Bike object) ==");
45         v2.start(); // Bike's overridden method
46         v2.stop(); // Bike's overridden method
47
48         System.out.println("\n== v3 (Vehicle object) ==");
49         v3.start(); // Vehicle's method
50         v3.stop(); // Vehicle's method
51     }
52 }
53 }
```

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 }
```

```
5     public Employee(String name, String id) {
6         this.name = name;
7         this.id = id;
8     }
9
10    public abstract double calculateSalary();
11
12    public void displayInfo() {
13        System.out.println("Name: " + name + ", ID: " + id +
14                                         ", Salary: $" + calculateSalary());
15    }
16 }
17
18 class Manager extends Employee {
19     private double baseSalary;
20     private double bonus;
21
22     public Manager(String name, String id, double baseSalary,
23                    double bonus) {
23         super(name, id);
24         this.baseSalary = baseSalary;
25         this.bonus = bonus;
26     }
27
28     @Override
29     public double calculateSalary() {
30         return baseSalary + bonus;
31     }
32 }
33
34 class Developer extends Employee {
35     private double hourlyRate;
36     private int hoursWorked;
37
38     public Developer(String name, String id, double hourlyRate,
39                      int hoursWorked) {
40         super(name, id);
41         this.hourlyRate = hourlyRate;
42         this.hoursWorked = hoursWorked;
43     }
44
45     @Override
46     public double calculateSalary() {
47         return hourlyRate * hoursWorked;
48     }
49 }
50
51 class Intern extends Employee {
52     private double stipend;
53
54     public Intern(String name, String id, double stipend) {
```

```
54     super(name, id);
55     this.stipend = stipend;
56 }
57
58 @Override
59 public double calculateSalary() {
60     return stipend;
61 }
62 }
63 public class Main {
64     public static void main(String[] args) {
65         // Polymorphic array - holds different employee types
66         Employee[] employees = new Employee[5];
67
68         employees[0] = new Manager("Alice", "M001", 8000, 2000);
69         employees[1] = new Developer("Bob", "D001", 50, 160);
70         employees[2] = new Developer("Charlie", "D002", 60, 150);
71         employees[3] = new Intern("David", "I001", 1000);
72         employees[4] = new Manager("Eve", "M002", 9000, 2500);
73
74         // Process all employees polymorphically
75         System.out.println("==== Employee Payroll ====");
76         double totalPayroll = 0;
77
78         for (Employee emp : employees) {
79             emp.displayInfo(); // Polymorphic call
80             totalPayroll += emp.calculateSalary();
81         }
82
83
84         System.out.println("\nTotal Payroll: $" + totalPayroll);
85
86         // Count employees by type
87         int managers = 0, developers = 0, interns = 0;
88
89         for (Employee emp : employees) {
90             if (emp instanceof Manager) {
91                 managers++;
92             } else if (emp instanceof Developer) {
93                 developers++;
94             } else if (emp instanceof Intern) {
95                 interns++;
96             }
97         }
98
99         System.out.println("\n==== Employee Count ====");
100        System.out.println("Managers: " + managers);
101        System.out.println("Developers: " + developers);
102        System.out.println("Interns: " + interns);
```

```
103     }
104 }
```

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:

```
1 // WITHOUT Polymorphism - Rigid and Hard to Maintain
2 class ShapeProcessor_Bad {
3     public void drawCircle(Circle circle) {
4         System.out.println("Drawing circle");
5     }
6
7     public void drawRectangle(Rectangle rectangle) {
8         System.out.println("Drawing rectangle");
9     }
10
11    public void drawTriangle(Triangle triangle) {
12        System.out.println("Drawing triangle");
13    }
14
15    // Need to add new method for every new shape!
16    // public void drawPentagon(Pentagon pentagon) { ... }
```

```
17 }
18 // WITH Polymorphism - Flexible and Easy to Extend
19 abstract class Shape {
20     abstract void draw();
21 }
22 }
23 class Circle extends Shape {
24     @Override
25     void draw() {
26         System.out.println("Drawing circle");
27     }
28 }
29 }
30 class Rectangle extends Shape {
31     @Override
32     void draw() {
33         System.out.println("Drawing rectangle");
34     }
35 }
36 }
37 class Triangle extends Shape {
38     @Override
39     void draw() {
40         System.out.println("Drawing triangle");
41     }
42 }
43 }
44 // Can add new shapes without changing this class!
45 class Pentagon extends Shape {
46     @Override
47     void draw() {
48         System.out.println("Drawing pentagon");
49     }
50 }
51 }
52 class ShapeProcessor_Good {
53     // One method handles ALL shapes!
54     public void drawShape(Shape shape) {
55         shape.draw(); // Polymorphic call
56     }
57 }
58 }
59 // Can process multiple shapes
60 public void drawAll(Shape[] shapes) {
61     for (Shape shape : shapes) {
62         shape.draw();
63     }
64 }
65 }
66 public class Main {
67     public static void main(String[] args) {
68         ShapeProcessor_Good processor = new
69             ShapeProcessor_Good();
70 }
```

```
71     Shape[] shapes = {  
72         new Circle(),  
73         new Rectangle(),  
74         new Triangle(),  
75         new Pentagon() // New shape - no changes needed!  
76     };  
77  
78     processor.drawAll(shapes);  
79 }  
80 }
```

Benefits:

- Code Flexibility
One method handles multiple types
- Code Extensibility
Add new types without changing existing code
- Code Maintainability
Changes in one place, not many
- Loose Coupling
Code depends on abstractions, not concrete implementations

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."

```
1 // Polymorphism with interfaces
2 interface Playable {
3     void play();
4 }
5 class Guitar implements Playable {
7     @Override
8     public void play() {
9         System.out.println("Playing guitar: Strum strum");
10    }
11 }
12 class Piano implements Playable {
14     @Override
15     public void play() {
16         System.out.println("Playing piano: Ding ding");
17    }
18 }
19 class Drums implements Playable {
21     @Override
22     public void play() {
23         System.out.println("Playing drums: Boom boom");
24    }
25 }
26 public class Main {
28     public static void playInstrument(Playable instrument) {
29         instrument.play(); // Polymorphism!
30     }
31
32     public static void main(String[] args) {
33         playInstrument(new Guitar());
34         playInstrument(new Piano());
35         playInstrument(new Drums());
36     }
37 }
```

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.

```
1 class Student {  
2     String name;  
3     int rollNumber;  
4  
5     // Constructor - same name as class, no return type  
6     Student() {  
7         System.out.println("Constructor called!");  
8         name = "Unknown";  
9         rollNumber = 0;  
10    }  
11 }  
12 public class Main {  
13     public static void main(String[] args) {  
14         Student s1 = new Student(); // Constructor is called  
15         here  
16         System.out.println("Name: " + s1.name);  
17         System.out.println("Roll Number: " + s1.rollNumber);  
18     }  
19 }
```

Output:

Constructor called!

Name: Unknown

Roll Number: 0

Student: What makes constructors special?

Expert: Let me show you the key characteristics:

```
1 class Person {
2     String name;
3     int age;
4
5     // ✅ Constructor characteristics:
6     // 1. Same name as class
7     // 2. No return type (not even void)
8     // 3. Called automatically when object is created
9     Person() {
10         System.out.println("Person constructor called");
11     }
12
13     // ❌ This is NOT a constructor (has return type)
14     void Person() {
15         System.out.println("This is a regular method, not a
constructor!");
16     }
17 }
18 public class Main {
19     public static void main(String[] args) {
20         Person p = new Person(); // Constructor called
21         p.Person(); // Regular method called
22     }
23 }
24 }
```

Types of Constructors

Student: Are there different types of constructors?

Expert: Yes! There are three main types:

1. Default Constructor (No-Argument)

```
1 class Book {
2     String title;
3     String author;
4     double price;
5
6     // Default constructor – no parameters
7     Book() {
8         System.out.println("Default constructor called");
9         title = "Unknown";
10        author = "Unknown";
11        price = 0.0;
12    }
13
14    void displayInfo() {
15        System.out.println("Title: " + title);
16        System.out.println("Author: " + author);
17        System.out.println("Price: $" + price);
18    }
19 }
20 public class Main {
21     public static void main(String[] args) {
22         Book book = new Book();
23         book.displayInfo();
24     }
25 }
26 }
```

Output:

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

2. Parameterized Constructor

```

1 class Book {
2     String title;
3     String author;
4     double price;
5
6     // Parameterized constructor
7     Book(String t, String a, double p) {
8         System.out.println("Parameterized constructor called");
9         title = t;
10        author = a;
11        price = p;
12    }
13
14    void displayInfo() {
15        System.out.println("Title: " + title);
16        System.out.println("Author: " + author);
17        System.out.println("Price: $" + price);
18    }
19 }
20 public class Main {
21     public static void main(String[] args) {
22         Book book = new Book("Java Programming", "John Doe",
23                             49.99);
24         book.displayInfo();
25     }
26 }
```

Output:

```

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

3. Copy Constructor

```

1 class Book {
2     String title;
3     String author;
4     double price;
```

```

5
6 // Parameterized constructor
7 Book(String title, String author, double price) {
8     this.title = title;
9     this.author = author;
10    this.price = price;
11 }
12
13 // Copy constructor - creates a copy of another object
14 Book(Book other) {
15     System.out.println("Copy constructor called");
16     this.title = other.title;
17     this.author = other.author;
18     this.price = other.price;
19 }
20
21 void displayInfo() {
22     System.out.println("Title: " + title + ", Author: " +
author + ", Price: $" + price);
23 }
24 }
25 public class Main {
26     public static void main(String[] args) {
27         Book original = new Book("Java Programming", "John Doe",
49.99);
28         Book copy = new Book(original); // Copy constructor
29
30         System.out.println("Original: ");
31         original.displayInfo();
32
33         System.out.println("\nCopy: ");
34         copy.displayInfo();
35
36         // Modify copy
37         copy.title = "Advanced Java";
38         copy.price = 59.99;
39
40         System.out.println("\nAfter modifying copy:");
41         System.out.println("Original: ");
42         original.displayInfo();
43
44         System.out.println("\nCopy: ");
45         copy.displayInfo();
46     }
47 }
48 }
```

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:

```
1 class Rectangle {
2     double length;
3     double width;
4
5     // Constructor 1: No parameters (default)
6     Rectangle() {
7         length = 1.0;
8         width = 1.0;
9         System.out.println("Default constructor: 1x1 rectangle
10        created");
11    }
12
13    // Constructor 2: One parameter (square)
14    Rectangle(double side) {
15        length = side;
16        width = side;
17        System.out.println("Square constructor: " + side + "x" +
18        side + " square created");
19    }
20
21    // Constructor 3: Two parameters (rectangle)
22    Rectangle(double length, double width) {
```

```
21         this.length = length;
22         this.width = width;
23         System.out.println("Rectangle constructor: " + length +
24             "x" + width + " rectangle created");
25     }
26
27     // Constructor 4: Copy constructor
28     Rectangle(Rectangle other) {
29         this.length = other.length;
30         this.width = other.width;
31         System.out.println("Copy constructor: copied rectangle");
32     }
33
34     double calculateArea() {
35         return length * width;
36     }
37
38     void displayInfo() {
39         System.out.println("Dimensions: " + length + " x " +
40             width);
41         System.out.println("Area: " + calculateArea());
42     }
43
44     public class Main {
45         public static void main(String[] args) {
46             Rectangle r1 = new Rectangle();
47             r1.displayInfo();
48             System.out.println();
49
50             Rectangle r2 = new Rectangle(5.0);
51             r2.displayInfo();
52             System.out.println();
53
54             Rectangle r3 = new Rectangle(4.0, 6.0);
55             r3.displayInfo();
56             System.out.println();
57
58             Rectangle r4 = new Rectangle(r3);
59             r4.displayInfo();
60         }
61     }
```

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()`:

```
1 class Employee {  
2     String name;  
3     int id;  
4     String department;  
5     double salary;  
6  
7     // Constructor 1: All parameters  
8     Employee(String name, int id, String department, double  
9         salary) {  
10        this.name = name;  
11        this.id = id;  
12        this.department = department;  
13        this.salary = salary;  
14        System.out.println("4-parameter constructor called");  
15    }  
16  
17    // Constructor 2: Calls Constructor 1 with default department  
18    Employee(String name, int id, double salary) {  
19        this(name, id, "General", salary); // Calls Constructor  
20    }
```

```

19         System.out.println("3-parameter constructor called");
20     }
21
22     // Constructor 3: Calls Constructor 2 with default salary
23     Employee(String name, int id) {
24         this(name, id, 30000.0); // Calls Constructor 2
25         System.out.println("2-parameter constructor called");
26     }
27
28     // Constructor 4: Calls Constructor 3 with default id
29     Employee(String name) {
30         this(name, 0); // Calls Constructor 3
31         System.out.println("1-parameter constructor called");
32     }
33
34     void displayInfo() {
35         System.out.println("Name: " + name);
36         System.out.println("ID: " + id);
37         System.out.println("Department: " + department);
38         System.out.println("Salary: $" + salary);
39     }
40 }
41 public class Main {
42     public static void main(String[] args) {
43         System.out.println("== Creating Employee 1 ==");
44         Employee emp1 = new Employee("Alice", 101, "IT", 50000);
45         emp1.displayInfo();
46
47         System.out.println("\n== Creating Employee 2 ==");
48         Employee emp2 = new Employee("Bob", 102, 45000);
49         emp2.displayInfo();
50
51         System.out.println("\n== Creating Employee 3 ==");
52         Employee emp3 = new Employee("Charlie", 103);
53         emp3.displayInfo();
54
55         System.out.println("\n== Creating Employee 4 ==");
56         Employee emp4 = new Employee("David");
57         emp4.displayInfo();
58     }
59 }
60 }
```

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:

```
1 class BadExample {  
2     // ✗ This will cause a compile error  
3     /*  
4     BadExample() {  
5         this(10); // Calls second constructor  
6     }  
7  
8     BadExample(int x) {  
9         this(); // Calls first constructor - CIRCULAR!  
10    }  
11    // Error: recursive constructor invocation  
12    */  
13 }
```

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:

```

1 // Example 1: No constructor written
2 class Student1 {
3     String name;
4     int age;
5
6     // Java automatically adds:
7     // Student1() { }
8 }
10 // Example 2: Constructor written
11 class Student2 {
12     String name;
13     int age;
14
15     Student2(String name, int age) {
16         this.name = name;
17         this.age = age;
18     }
19
20     // Java does NOT add default constructor
21 }
22 public class Main {
23     public static void main(String[] args) {
24         // ✓ Works - Java provided default constructor
25         Student1 s1 = new Student1();
26
27         // ✓ Works - using our constructor
28         Student2 s2 = new Student2("Alice", 20);
29
30         // ✗ Compile error - no default constructor in Student2
31         // Student2 s3 = new Student2();
32     }
33 }
34 }
```

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

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

```

1 class BankAccount {
2     String accountNumber;
3     double balance;
4
5     // We added a parameterized constructor
6     BankAccount(String accountNumber, double balance) {
7         this.accountNumber = accountNumber;
8         this.balance = balance;
9     }
10 }
11 class SavingsAccount extends BankAccount {
12     double interestRate;
13
14
15     SavingsAccount(String accountNumber, double balance, double
16     interestRate) {
17         // ❌ This will cause an error!
18         // Java tries to call super() automatically
19         // But BankAccount doesn't have a no-arg constructor
20         anymore!
21         // ✅ Fix: explicitly call parent's parameterized
22         constructor
23         super(accountNumber, balance);
24         this.interestRate = interestRate;
25     }
26 }
```

Private Constructors

Student: Can constructors be private?

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

```

1 // Use Case 1: Singleton Pattern
2 class DatabaseConnection {
3     private static DatabaseConnection instance;
4     private String connectionString;
5
6     // Private constructor - cannot create objects from outside
7     private DatabaseConnection() {
8         connectionString = "jdbc:mysql://localhost:3306/mydb";
9         System.out.println("Database connection created");
10    }
```

```
11
12     // Public method to get the single instance
13     public static DatabaseConnection getInstance() {
14         if (instance == null) {
15             instance = new DatabaseConnection();
16         }
17         return instance;
18     }
19
20     public void connect() {
21         System.out.println("Connecting to: " + connectionString);
22     }
23 }
25 // Use Case 2: Utility Class (only static methods)
26 class MathUtils {
27     // Private constructor - prevents instantiation
28     private MathUtils() {
29         throw new AssertionError("Cannot instantiate utility
30         class");
31     }
32     public static int add(int a, int b) {
33         return a + b;
34     }
35
36     public static int multiply(int a, int b) {
37         return a * b;
38     }
39 }
40 public class Main {
41     public static void main(String[] args) {
42         // ✗ Cannot do this - constructor is private
43         // DatabaseConnection db = new DatabaseConnection();
44
45         // ✓ Get singleton instance
46         DatabaseConnection db1 =
47             DatabaseConnection.getInstance();
48         DatabaseConnection db2 =
49             DatabaseConnection.getInstance();
50
51         System.out.println("db1 == db2: " + (db1 == db2));    // true - same instance
52         db1.connect();
53
54         // ✗ Cannot do this - constructor is private
55         // MathUtils utils = new MathUtils();
56 }
```

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

```

1 class Demo {
2     String name;
3
4     // CONSTRUCTOR
5     Demo(String name) {
6         this.name = name;
7         System.out.println("Constructor called");
8     }
9
10    // REGULAR METHOD
11    void setName(String name) {
12        this.name = name;
13        System.out.println("Method called");
14    }
15 }
16 public class Main {
17     public static void main(String[] args) {
18         Demo obj = new Demo("Initial"); // Constructor called
automatically
19         obj.setName("Updated"); // Method called
explicitly
20     }
21 }
22 }
```

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:

```
1 class Car {  
2     // Required fields  
3     private String brand;  
4     private String model;  
5     private int year;  
6  
7     // Optional fields  
8     private String color;  
9     private String engineType;  
10    private boolean hasSunroof;  
11    private boolean hasLeatherSeats;  
12    private boolean hasNavigationSystem;  
13  
14    // Constructor 1: Minimum required fields  
15    public Car(String brand, String model, int year) {  
16        this.brand = brand;  
17        this.model = model;  
18        this.year = year;  
19        // Optional fields get default values  
20        this.color = "White";  
21        this.engineType = "Petrol";  
22        this.hasSunroof = false;  
23        this.hasLeatherSeats = false;  
24        this.hasNavigationSystem = false;  
25        System.out.println("Basic car created");  
26    }  
27  
28    // Constructor 2: With color  
29    public Car(String brand, String model, int year, String  
30    color) {  
31        this(brand, model, year); // Call Constructor 1  
32        this.color = color;  
33        System.out.println("Car with custom color created");  
34    }  
35  
36    // Constructor 3: With color and engine type  
37    public Car(String brand, String model, int year, String  
38    color, String engineType) {  
39        this(brand, model, year, color); // Call Constructor 2  
40        this.engineType = engineType;
```

```
39         System.out.println("Car with custom engine created");
40     }
41
42     // Constructor 4: Full customization
43     public Car(String brand, String model, int year, String
44     color, String engineType,
45             boolean hasSunroof, boolean hasLeatherSeats,
46     boolean hasNavigationSystem) {
47         this(brand, model, year, color, engineType); // Call
48         Constructor 3
49         this.hasSunroof = hasSunroof;
50         this.hasLeatherSeats = hasLeatherSeats;
51         this.hasNavigationSystem = hasNavigationSystem;
52         System.out.println("Fully customized car created");
53     }
54
55     // Copy constructor
56     public Car(Car other) {
57         this.brand = other.brand;
58         this.model = other.model;
59         this.year = other.year;
60         this.color = other.color;
61         this.engineType = other.engineType;
62         this.hasSunroof = other.hasSunroof;
63         this.hasLeatherSeats = other.hasLeatherSeats;
64         this.hasNavigationSystem = other.hasNavigationSystem;
65         System.out.println("Car copied");
66     }
67
68     public void displaySpecs() {
69         System.out.println("==> Car Specifications ==>");
70         System.out.println("Brand: " + brand);
71         System.out.println("Model: " + model);
72         System.out.println("Year: " + year);
73         System.out.println("Color: " + color);
74         System.out.println("Engine: " + engineType);
75         System.out.println("Sunroof: " + (hasSunroof ? "Yes" :
76             "No"));
77         System.out.println("Leather Seats: " + (hasLeatherSeats ?
78             "Yes" : "No"));
79         System.out.println("Navigation: " + (hasNavigationSystem
80             ? "Yes" : "No"));
81         System.out.println("Price: $" + calculatePrice());
82         System.out.println();
83     }
84
85     private double calculatePrice() {
86         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",
111                            "Electric",
112                            true, true, true);
113        car4.displaySpecs();
114
115        // Copy car
116        System.out.println("==> Car 5 (Copy of Car 4) ==>");
117        Car car5 = new Car(car4);
118        car5.displaySpecs();
119    }
120 }

```

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."

```

1 class Parent {
2     int x;
3
4     Parent(int x) {
5         this.x = x;
6         System.out.println("Parent constructor");
7     }
8 }
10 class Child extends Parent {
11     int y;
12
13     // Child does NOT inherit Parent's constructor
14     // Must explicitly call it
15     Child(int x, int y) {
16         super(x); // Must call parent constructor
17         this.y = y;
18         System.out.println("Child constructor");
19     }
20 }
22 public class Main {
23     public static void main(String[] args) {
24         Child c = new Child(10, 20);
25     }
26 }
```

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:

```
1 class Person {  
2     String name;  
3     int age;  
4  
5     void setDetails(String name, int age) {  
6         // Without 'this', this is confusing:  
7         // name = name; // Which name? Parameter or field?  
8  
9         // With 'this', it's clear:  
10        this.name = name; // this.name is the field, name is the  
11          parameter  
12        this.age = age;  
13    }  
14  
15    void display() {  
16        System.out.println("Name: " + this.name); // 'this' is  
17          optional here  
18        System.out.println("Age: " + this.age);  
19    }  
20}  
21 public class Main {  
22     public static void main(String[] args) {  
23         Person p = new Person();  
24         p.setDetails("Alice", 25);  
25         p.display();  
26     }  
27 }
```

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

```
1 class Student {  
2     String name;  
3     int rollNumber;  
4  
5     // Without 'this' - CONFUSING  
6     void setDetails_Bad(String name, int rollNumber) {  
7         name = name; // Assigns parameter to itself!  
8         rollNumber = rollNumber; // Assigns parameter to itself!  
9         // Instance variables remain unchanged!  
10    }  
11  
12    // With 'this' - CLEAR  
13    void setDetails_Good(String name, int rollNumber) {  
14        this.name = name; // Assigns parameter to instance  
variable  
15        this.rollNumber = rollNumber;  
16    }  
17  
18    void display() {  
19        System.out.println("Name: " + name + ", Roll: " +  
rollNumber);  
20    }  
21 }  
22 public class Main {  
23     public static void main(String[] args) {  
24         Student s1 = new Student();  
25         s1.setDetails_Bad("Alice", 101);  
26         s1.display(); // Name: null, Roll: 0 (not set!)  
27  
28         Student s2 = new Student();  
29         s2.setDetails_Good("Bob", 102);  
30         s2.display(); // Name: Bob, Roll: 102 (correctly set!)  
31     }  
32 }  
33 }
```

Use 2: Call Current Class Method

```

1 class Calculator {
2     int result = 0;
3
4     void add(int a, int b) {
5         result = a + b;
6         this.display(); // Calling current class method
7         // display(); would also work, but 'this' makes it
explicit
8     }
9
10    void subtract(int a, int b) {
11        result = a - b;
12        this.display();
13    }
14
15    void display() {
16        System.out.println("Result: " + result);
17    }
18 }
19 public class Main {
20     public static void main(String[] args) {
21         Calculator calc = new Calculator();
22         calc.add(10, 5);
23         calc.subtract(10, 5);
24     }
25 }
26 }
```

Use 3: Call Current Class Constructor (Constructor Chaining)

```

1 class Employee {
2     String name;
3     int id;
4     String department;
5     double salary;
6
7     // Constructor 1
8     Employee() {
9         this("Unknown", 0, "General", 0.0); // Calls Constructor
10
11         System.out.println("Default constructor");
12     }
13
14     // Constructor 2
15     Employee(String name, int id) {
```

```

15         this(name, id, "General", 30000.0); // Calls Constructor
4
16         System.out.println("2-parameter constructor");
17     }
18
19     // Constructor 3
20     Employee(String name, int id, String department) {
21         this(name, id, department, 30000.0); // Calls
22         Constructor 4
23         System.out.println("3-parameter constructor");
24     }
25
26     // Constructor 4
27     Employee(String name, int id, String department, double
28     salary) {
29         this.name = name;
30         this.id = id;
31         this.department = department;
32         this.salary = salary;
33         System.out.println("4-parameter constructor");
34     }
35
36     void display() {
37         System.out.println(name + " | " + id + " | " + department
38         + " | $" + salary);
39     }
40
41 public class Main {
42     public static void main(String[] args) {
43         Employee e1 = new Employee();
44         e1.display();
45
46         System.out.println();
47         Employee e2 = new Employee("Alice", 101);
48         e2.display();
49     }

```

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

```
1 class Student {
2     String name;
3     int marks;
4
5     Student(String name, int marks) {
6         this.name = name;
7         this.marks = marks;
8     }
9
10    void compareWith(Student other) {
11        if (this.marks > other.marks) {
12            System.out.println(this.name + " scored more than " +
other.name);
13        } else if (this.marks < other.marks) {
14            System.out.println(this.name + " scored less than " +
other.name);
15        } else {
16            System.out.println(this.name + " and " + other.name +
" scored equal marks");
17        }
18    }
19
20    void printDetails() {
21        // Passing 'this' as parameter to another class method
22        Printer.print(this);
23    }
24 }
25 class Printer {
26     static void print(Student student) {
27         System.out.println("Student: " + student.name + ", Marks:
" + student.marks);
28     }
29 }
30 }
```

```
32 public class Main {  
33     public static void main(String[] args) {  
34         Student s1 = new Student("Alice", 85);  
35         Student s2 = new Student("Bob", 90);  
36  
37         s1.compareWith(s2);  
38         s2.compareWith(s1);  
39  
40         System.out.println();  
41  
42         s1.printDetails();  
43         s2.printDetails();  
44     }  
45 }
```

Output:

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

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

Use 5: Return Current Object

```

1 class Builder {
2     private String name;
3     private int age;
4     private String city;
5
6     // Method chaining using 'this'
7     Builder setName(String name) {
8         this.name = name;
9         return this; // Returns current object
10    }
11
12    Builder setAge(int age) {
13        this.age = age;
14        return this; // Returns current object
15    }
16
17    Builder setCity(String city) {
18        this.city = city;
19        return this; // Returns current object
20    }
21
22    void display() {
23        System.out.println("Name: " + name + ", Age: " + age + ", "
24            City: " + city);
25    }
26 }
27 public class Main {
28     public static void main(String[] args) {
29         Builder builder = new Builder();
30
31         // Method chaining – possible because each method returns
'this'
32         builder.setName("Alice")
33             .setAge(25)
34             .setCity("New York")
35             .display();
36
37         // Same as:
38         // builder.setName("Alice");
39         // builder.setAge(25);
40         // builder.setCity("New York");
41         // builder.display();
42     }
43 }

```

Output:

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

Use 6: Pass as Argument in Constructor Call

```
1 class Address {  
2     String city;  
3     String country;  
4  
5     Address(String city, String country) {  
6         this.city = city;  
7         this.country = country;  
8     }  
9  
10    void display() {  
11        System.out.println("Address: " + city + ", " + country);  
12    }  
13 }  
14 class Person {  
15     String name;  
16     Address address;  
17  
18     Person(String name, String city, String country) {  
19         this.name = name;  
20         // Passing 'this' is not needed here, but showing the  
21         // concept  
22         this.address = new Address(city, country);  
23     }  
24  
25     void display() {  
26         System.out.println("Person: " + name);  
27         address.display();  
28     }  
29 }  
30 // Better example showing 'this' in constructor  
31 class Employee {  
32     String name;  
33     EmployeeDetails details;  
34  
35     Employee(String name, int id) {  
36         this.name = name;  
37         // Passing current Employee object to EmployeeDetails  
38         this.details = new EmployeeDetails(this, id);  
39     }  
40 }
```

```

41
42     void display() {
43         System.out.println("Employee: " + name);
44         details.display();
45     }
46 }
48 class EmployeeDetails {
49     Employee employee; // Reference to Employee object
50     int id;
51
52     EmployeeDetails(Employee employee, int id) {
53         this.employee = employee;
54         this.id = id;
55     }
56
57     void display() {
58         System.out.println("ID: " + id + ", Name: " +
employee.name);
59     }
60 }
62 public class Main {
63     public static void main(String[] args) {
64         Employee emp = new Employee("Alice", 101);
65         emp.display();
66     }
67 }
```

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;
```

```
5     private String accountType;
6
7     // Use 1: Distinguish parameters from instance variables
8     public BankAccount(String accountNumber, String holderName,
9                         double balance, String accountType) {
10            this.accountNumber = accountNumber;
11            this.holderName = holderName;
12            this.balance = balance;
13            this.accountType = accountType;
14        }
15
16        // Use 3: Constructor chaining
17        public BankAccount(String accountNumber, String holderName) {
18            this(accountNumber, holderName, 0.0, "Savings"); // Calls main constructor
19        }
20
21        // Use 5: Return current object for method chaining
22        public BankAccount deposit(double amount) {
23            if (amount > 0) {
24                this.balance += amount;
25                System.out.println("Deposited: $" + amount);
26                this.printBalance(); // Use 2: Call current class method
27            }
28            return this; // Return current object
29        }
30
31        public BankAccount withdraw(double amount) {
32            if (amount > 0 && amount <= this.balance) {
33                this.balance -= amount;
34                System.out.println("Withdrawn: $" + amount);
35                this.printBalance();
36            } else {
37                System.out.println("Insufficient balance or invalid amount!");
38            }
39            return this; // Return current object
40        }
41
42        // Use 4: Pass current object as parameter
43        public void transferTo(BankAccount recipient, double amount)
44        {
45            if (amount > 0 && amount <= this.balance) {
46                this.balance -= amount;
47                recipient.balance += amount;
48                System.out.println("Transferred $" + amount + " from " +
49                    this.holderName + " to " + recipient.holderName);
50            }
51        }
52    }
53}
```

```
47                         this.holderName + " to " +
recipient.holderName);
48                     this.printBalance();
49                     recipient.printBalance();
50                 } else {
51                     System.out.println("Transfer failed!");
52                 }
53             }
54
55     public void compareBalance(BankAccount other) {
56         if (this.balance > other.balance) {
57             System.out.println(this.holderName + " has more
balance than " + other.holderName);
58         } else if (this.balance < other.balance) {
59             System.out.println(this.holderName + " has less
balance than " + other.holderName);
60         } else {
61             System.out.println(this.holderName + " and " +
other.holderName + " have equal balance");
62         }
63     }
64
65     private void printBalance() {
66         System.out.println(this.holderName + "'s balance: $" +
this.balance);
67     }
68
69     public void displayInfo() {
70         System.out.println("== Account Information ==");
71         System.out.println("Account Number: " +
this.accountNumber);
72         System.out.println("Holder Name: " + this.holderName);
73         System.out.println("Account Type: " + this.accountType);
74         System.out.println("Balance: $" + this.balance);
75         System.out.println();
76     }
77 }
78 public class Main {
79     public static void main(String[] args) {
80         // Using constructor chaining
81         BankAccount account1 = new BankAccount("ACC001",
82                                         "Alice");
83         BankAccount account2 = new BankAccount("ACC002", "Bob",
84                                         5000, "Current");
85         account1.displayInfo();
86         account2.displayInfo();
87 }
```

```
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

```

1 class Demo {
2     static int staticVar = 10;
3     int instanceVar = 20;
4
5     static void staticMethod() {
6         System.out.println("Static var: " + staticVar); // ✓
7         OK
8         // ✗ Cannot use 'this' in static context
9         // System.out.println("Instance var: " +
10            this.instanceVar); // Compile error!
11        Why? 'this' refers to current object, but static
12        methods
13    }
14
15    void instanceMethod() {
16        System.out.println("Static var: " + staticVar); // ✓
17        OK
18        System.out.println("Instance var: " + this.instanceVar);
19        // ✓ OK
20    }
21 }
22 public class Main {
23     public static void main(String[] args) {
24         Demo.staticMethod(); // Called on class
25         Demo obj = new Demo();
26         obj.instanceMethod(); // Called on object
27     }
28 }
```

Mistake 2: Forgetting **this** When Needed

```
1 class Person {
2     String name;
3
4     // ❌ Bug: Parameter shadows instance variable
5     void setName(String name) {
6         name = name; // Assigns parameter to itself!
7         // Instance variable 'name' remains unchanged
8     }
9
10    // ✅ Correct: Use 'this' to access instance variable
11    void setNameCorrect(String name) {
12        this.name = name;
13    }
14
15    void display() {
16        System.out.println("Name: " + name);
17    }
18 }
19
20 public class Main {
21     public static void main(String[] args) {
22         Person p1 = new Person();
23         p1.setName("Alice");
24         p1.display(); // Name: null (Bug!)
25
26         Person p2 = new Person();
27         p2.setNameCorrect("Bob");
28         p2.display(); // Name: Bob (Correct!)
29     }
30 }
```

Mistake 3: Circular Reference with **this()**

```
1 class BadExample {
2     // X This causes infinite recursion
3     /*
4     BadExample() {
5         this(10); // Calls second constructor
6     }
7
8     BadExample(int x) {
9         this(); // Calls first constructor - CIRCULAR!
10    }
11    // Compile error: recursive constructor invocation
12    */
13 }
```

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

```
1 class WrongConstructor {
2     int x;
3
4     // X this() must be first statement
5     /*
6     WrongConstructor(int x) {
7         this.x = x;
8         this(); // Compile error: call to this must be first
9             statement
10    }
11
12    WrongConstructor() {
13        System.out.println("Default constructor");
14    }
15    */
16    // ✓ Correct way
17    WrongConstructor(int x) {
18        this(); // Must be first
19        this.x = x;
20    }
21
22    WrongConstructor() {
23        System.out.println("Default constructor");
24    }
25 }
```

this vs super

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

Expert: Great question! Let me show you:

```
1 class Parent {
2     String name = "Parent";
3
4     Parent() {
5         System.out.println("Parent constructor");
6     }
7
8     Parent(String name) {
9         this.name = name;
10        System.out.println("Parent parameterized constructor");
11    }
12
13    void display() {
14        System.out.println("Parent display: " + name);
15    }
16 }
17
18 class Child extends Parent {
19     String name = "Child";
20
21     Child() {
22         super(); // Calls Parent's no-arg constructor
23         System.out.println("Child constructor");
24     }
25
26     Child(String parentName, String childName) {
27         super(parentName); // Calls Parent's parameterized
28         // constructor
29         this.name = childName; // 'this' refers to Child's name
30         System.out.println("Child parameterized constructor");
31     }
32
33     void display() {
34         System.out.println("Child display: " + name);
35     }
36
37     void showBoth() {
38         System.out.println("Child name (this): " + this.name);
39         System.out.println("Parent name (super): " + super.name);
40     }
41 }
```

```

41     void callBothDisplays() {
42         this.display();    // Calls Child's display
43         super.display(); // Calls Parent's display
44     }
45 }
46 public class Main {
47     public static void main(String[] args) {
48         System.out.println("== Creating Child with default
49         constructor ==");
50         Child c1 = new Child();
51         c1.showBoth();
52
53         System.out.println("\n== Creating Child with
54         parameterized constructor ==");
55         Child c2 = new Child("CustomParent", "CustomChild");
56         c2.showBoth();
57
58         System.out.println("\n== Calling both displays ==");
59         c2.callBothDisplays();
60     }
61 }

```

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 this.method() this()	super.variable super.method() 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."

```

1 class Parent {
2     Parent() {
3         System.out.println("Parent constructor");
4     }
5 }
6 class Child extends Parent {
7     // X Cannot have both
8     /*
9      Child() {
10         super(); // First statement
11         this(10); // Also wants to be first statement - ERROR!
12     }
13 */
14
15
16 // ✓ Choose one
17 Child() {
18     super(); // Calls parent constructor
19     System.out.println("Child no-arg constructor");
20 }
21
22 Child(int x) {
23     this(); // Calls Child's no-arg constructor (which calls
24     super)
25     System.out.println("Child parameterized constructor");
26 }

```

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."

```

1 class Demo {
2     static int staticVar = 10;
3     int instanceVar = 20;
4
5     void display() {
6         // ✓ Works but not recommended
7         System.out.println(this.staticVar);
8
9         // ✓ Recommended way
10        System.out.println(Demo.staticVar);
11
12        // ✓ This is fine
13        System.out.println(this.instanceVar);
14    }
15 }

```

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  
4         class  
5     protected int protectedVar = 3;    // Accessible in this class  
6         and subclasses  
7     int defaultVar = 4;               // Accessible in same  
8         package (no keyword)  
9 }
```

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:

```

1 // BAD: Everything is public
2 class BankAccount_Bad {
3     public double balance; // Anyone can modify!
4
5     public void withdraw(double amount) {
6         balance -= amount;
7     }
8 }
10 // GOOD: Proper encapsulation
11 class BankAccount_Good {
12     private double balance; // Hidden from outside
13
14     public void withdraw(double amount) {
15         if (amount > 0 && amount <= balance) {
16             balance -= amount;
17         } else {
18             System.out.println("Invalid withdrawal!");
19         }
20     }
21
22     public double getBalance() {
23         return balance;
24     }
25 }
26 public class Main {
27     public static void main(String[] args) {
28         BankAccount_Bad bad = new BankAccount_Bad();
29         bad.balance = 1000;
30         bad.balance = -5000; // ✗ Can set negative balance!
31
32         BankAccount_Good good = new BankAccount_Good();
33         // good.balance = 1000; // ✗ Compile error – cannot
34         // access private field
35         good.withdraw(-5000); // ✓ Validation prevents this
36     }
37 }

```

The Four Access Modifiers

Expert: Java has four access modifiers. Let me explain each:

1. Public

```
1 // Accessible from ANYWHERE
2 public class PublicDemo {
3     public int publicVar = 10;
4
5     public void publicMethod() {
6         System.out.println("Public method - accessible
everywhere");
7     }
8 }
10 // Can be accessed from any class, any package
11 class AnyClass {
12     void test() {
13         PublicDemo demo = new PublicDemo();
14         System.out.println(demo.publicVar); // ✓ OK
15         demo.publicMethod(); // ✓ OK
16     }
17 }
```

2. Private

```
1 // Accessible ONLY within the same class
2 class PrivateDemo {
3     private int privateVar = 20;
4
5     private void privateMethod() {
6         System.out.println("Private method - only accessible in
this class");
7     }
8
9     public void publicMethod() {
10        // Can access private members within same class
11        System.out.println(privateVar); // ✓ OK
12        privateMethod(); // ✓ OK
13    }
14 }
15 class AnotherClass {
16     void test() {
17         PrivateDemo demo = new PrivateDemo();
18         // System.out.println(demo.privateVar); // ✗ Compile
error
19         // demo.privateMethod(); // ✗ Compile error
20         demo.publicMethod(); // ✓ OK
21     }
22 }
23 }
```

3. Protected

```

1 // Accessible within same package and subclasses
2 class ProtectedDemo {
3     protected int protectedVar = 30;
4
5     protected void protectedMethod() {
6         System.out.println("Protected method");
7     }
8 }
10 // Same package - can access
11 class SamePackageClass {
12     void test() {
13         ProtectedDemo demo = new ProtectedDemo();
14         System.out.println(demo.protectedVar); // ✓ OK (same
package)
15         demo.protectedMethod(); // ✓ OK (same package)
16     }
17 }
18 // Different package but subclass - can access
19 class SubclassInDifferentPackage extends ProtectedDemo {
20     void test() {
21         System.out.println(protectedVar); // ✓ OK (inherited)
22         protectedMethod(); // ✓ OK (inherited)
23     }
24 }
25 }
26 // Different package, not subclass - cannot access
27 class DifferentPackageClass {
28     void test() {
29         ProtectedDemo demo = new ProtectedDemo();
30         // System.out.println(demo.protectedVar); // ✗ Compile
error
31         // demo.protectedMethod(); // ✗ Compile error
32     }
33 }
34 }

```

4. Default (Package-Private)

```

1 // Accessible only within the same package (no keyword)
2 class DefaultDemo {
3     int defaultVar = 40; // No access modifier = default
4
5     void defaultMethod() {
6         System.out.println("Default method");
7     }
8 }
10 // Same package - can access
11 class SamePackageClass {
12     void test() {
13         DefaultDemo demo = new DefaultDemo();
14         System.out.println(demo.defaultVar); // ✓ OK
15         demo.defaultMethod(); // ✓ OK
16     }
17 }
18 // Different package - cannot access
20 /*
21 package other;
23 class DifferentPackageClass {
24     void test() {
25         DefaultDemo demo = new DefaultDemo();
26         // System.out.println(demo.defaultVar); // ✗ Compile
error
27         // demo.defaultMethod(); // ✗ Compile error
28     }
29 }
30 */

```

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:

```
1 class AccessModifierDemo {  
2     // Different access levels  
3     public int publicVar = 1;  
4     protected int protectedVar = 2;  
5     int defaultVar = 3; // package-private  
6     private int privateVar = 4;  
7  
8     // Public method - accessible everywhere  
9     public void publicMethod() {  
10         System.out.println("Public method called");  
11         // Can access all variables within same class  
12         System.out.println("Public: " + publicVar);  
13         System.out.println("Protected: " + protectedVar);  
14         System.out.println("Default: " + defaultVar);  
15         System.out.println("Private: " + privateVar);  
16     }  
17  
18     // Protected method - accessible in package and subclasses  
19     protected void protectedMethod() {  
20         System.out.println("Protected method called");  
}
```

```

21     }
22
23     // Default method - accessible only in same package
24     void defaultMethod() {
25         System.out.println("Default method called");
26     }
27
28     // Private method - accessible only in this class
29     private void privateMethod() {
30         System.out.println("Private method called");
31     }
32
33     // Public method to demonstrate private method access
34     public void callPrivateMethod() {
35         privateMethod(); // Can call private method within same
36         class
37     }
38 // Same package class
39 class SamePackageTest {
40     void test() {
41         AccessModifierDemo demo = new AccessModifierDemo();
42
43         System.out.println("==== Same Package Access ====");
44         System.out.println("Public var: " + demo.publicVar);
45         // ✅ OK
46         System.out.println("Protected var: " +
47             demo.protectedVar); // ✅ OK
48         System.out.println("Default var: " + demo.defaultVar);
49         // ✅ OK
50         // System.out.println("Private var: " + demo.privateVar);
51         // ✗ Error
52         demo.publicMethod(); // ✅ OK
53         demo.protectedMethod(); // ✅ OK
54         demo.defaultMethod(); // ✅ OK
55         // demo.privateMethod(); // ✗ Error
56         demo.callPrivateMethod(); // ✅ OK (calls private
57         method indirectly)
58     }
59 // Subclass in same package
60 class SubclassInSamePackage extends AccessModifierDemo {
61     void test() {
62         System.out.println("==== Subclass in Same Package ====");
63         System.out.println("Public var: " + publicVar); // ✅ OK (inherited)
64         System.out.println("Protected var: " + protectedVar); // ✅ OK

```

```

    ✓ OK (inherited)
64      System.out.println("Default var: " + defaultVar);      //
    ✓ OK (inherited)
65      // System.out.println("Private var: " + privateVar);  //
✗ Error (not inherited)

66
67      publicMethod();      // ✓ OK
68      protectedMethod();   // ✓ OK
69      defaultMethod();    // ✓ OK
70      // privateMethod(); // ✗ Error
71  }
72 }

73 public class Main {
74     public static void main(String[] args) {
75         SamePackageTest test1 = new SamePackageTest();
76         test1.test();
77
78         System.out.println();
79
80         SubclassInSamePackage test2 = new
81             SubclassInSamePackage();
82         test2.test();
83     }
84 }
```

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:

```

1 class Employee {
2     // Private fields - hidden from outside
3     private String employeeId;
4     private String name;
5     private double salary;
6     private String department;
7
8     // Protected field - accessible to subclasses
9     protected int yearsOfExperience;
10
11    // Package-private field - accessible in same package
12    String officeLocation;
13
14    // Public constructor
```

```
15     public Employee(String employeeId, String name, double
16         salary, String department) {
17         this.employeeId = employeeId;
18         this.name = name;
19         this.salary = salary;
20         this.department = department;
21         this.yearsOfExperience = 0;
22         this.officeLocation = "Head Office";
23     }
24
25     // Public getters - controlled read access
26     public String getEmployeeId() {
27         return employeeId;
28     }
29
30     public String getName() {
31         return name;
32     }
33
34     public String getDepartment() {
35         return department;
36     }
37
38     // Public setter with validation
39     public void setName(String name) {
40         if (name != null && !name.trim().isEmpty()) {
41             this.name = name;
42         }
43     }
44
45     // Protected method - accessible to subclasses
46     protected double getSalary() {
47         return salary;
48     }
49
50     // Protected method with validation
51     protected void setSalary(double salary) {
52         if (salary > 0) {
53             this.salary = salary;
54         }
55     }
56
57     // Private helper method - internal use only
58     private double calculateTax() {
59         return salary * 0.2; // 20% tax
60     }
61
62     // Private helper method
```

```

62     private double calculateBonus() {
63         return salary * 0.1 * yearsOfExperience;
64     }
65
66     // Public method using private methods
67     public double getNetSalary() {
68         return salary - calculateTax() + calculateBonus();
69     }
70
71     // Package-private method
72     void updateOfficeLocation(String location) {
73         this.officeLocation = location;
74     }
75
76     // Public method
77     public void displayInfo() {
78         System.out.println("==== Employee Information ====");
79         System.out.println("ID: " + employeeId);
80         System.out.println("Name: " + name);
81         System.out.println("Department: " + department);
82         System.out.println("Office: " + officeLocation);
83         System.out.println("Experience: " + yearsOfExperience + " "
84             + years);
85         System.out.println("Gross Salary: $" + salary);
86         System.out.println("Tax: $" + calculateTax());
87         System.out.println("Bonus: $" + calculateBonus());
88         System.out.println("Net Salary: $" + getNetSalary());
89     }
90     class Manager extends Employee {
91         private int teamSize;
92
93         public Manager(String employeeId, String name, double salary,
94             String department, int teamSize) {
95             super(employeeId, name, salary, department);
96             this.teamSize = teamSize;
97         }
98
99         // Can access protected members from parent
100        public void giveRaise(double percentage) {
101            double currentSalary = getSalary(); // ✓ OK –
102            protected method
103            double newSalary = currentSalary * (1 + percentage / 100);
104            setSalary(newSalary); // ✓ OK – protected method
105            System.out.println("Raise given! New salary: $" +
newSalary);
106        }

```

```

106
107     public void addExperience(int years) {
108         yearsOfExperience += years; // ✓ OK - protected field
109     }
110
111     @Override
112     public void displayInfo() {
113         super.displayInfo();
114         System.out.println("Team Size: " + teamSize);
115     }
116 }
118 // Same package class - HR Department
119 class HRDepartment {
120     public void processEmployee(Employee emp) {
121         System.out.println("== HR Processing ==");
122         System.out.println("Employee: " + emp.getName()); // ✓
123         System.out.println("Department: " + emp.getDepartment()); // ✓
124         System.out.println("Office: " + emp.officeLocation); // ✓
125         // Can update office location (package-private method)
126         emp.updateOfficeLocation("Branch Office"); // ✓ OK
127
128         // Cannot access private or protected members
129         // System.out.println("Salary: " + emp.getSalary()); // ✗ Error - protected
130         // System.out.println("Salary: " + emp.salary); // ✗ Error - private
131     }
132 }
133 }
135 public class Main {
136     public static void main(String[] args) {
137         Employee emp = new Employee("E001", "Alice", 50000,
138             "IT");
139         emp.yearsOfExperience = 5; // ✓ OK - protected (same
140         package)
141         emp.displayInfo();
142
143         System.out.println();
144
145         Manager mgr = new Manager("M001", "Bob", 80000,
146             "Management", 10);
147         mgr.addExperience(10);
148         mgr.giveRaise(10);
149         mgr.displayInfo();
150
151     }
152 }
```

```
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:

```
1 //  Public class - accessible from anywhere
2 // File name MUST match class name (PublicClass.java)
3 public class PublicClass {
4     public void display() {
5         System.out.println("Public class");
6     }
7 }
8 //  Default class - accessible only in same package
9 // Can have any file name
10 class DefaultClass {
11     void display() {
12         System.out.println("Default class");
13     }
14 }
15 }
16 //  Cannot have protected or private top-level classes
17 // protected class ProtectedClass {} // Compile error
18 // private class PrivateClass {} // Compile error
19 //  But can have private/protected nested classes
20 class OuterClass {
21     private class PrivateInnerClass {
22         void display() {
23             System.out.println("Private inner class");
24         }
25     }
26 }
27 }
28
29     protected class ProtectedInnerClass {
30         void display() {
31             System.out.println("Protected inner class");
32         }
33     }
34
35     public void test() {
36         PrivateInnerClass pic = new PrivateInnerClass();
37         pic.display(); //  OK - same outer class
38     }
39 }
```

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:

```
1 class BestPracticesDemo {  
2     //  RULE 1: Make fields private by default  
3     private String name;  
4     private int age;  
5     private double salary;  
6  
7     //  RULE 2: Provide public getters/setters only when  
8     // needed  
9     public String getName() {  
10        return name;  
11    }  
12  
13    public void setName(String name) {  
14        if (name != null && !name.trim().isEmpty()) {  
15            this.name = name;  
16        }  
17    }  
18  
19    //  RULE 3: Read-only fields – getter but no setter  
20    public int getAge() {
```

```
20         return age;
21     }
22
23     // ✅ RULE 4: No getter for sensitive data
24     // No getSalary() method – salary is completely hidden
25
26     // ✅ RULE 5: Use protected for inheritance-specific members
27     protected void calculateBonus() {
28         // Subclasses can override this
29     }
30
31     // ✅ RULE 6: Private helper methods
32     private boolean isValidAge(int age) {
33         return age >= 18 && age <= 65;
34     }
35
36     // ✅ RULE 7: Public API methods
37     public void updateAge(int newAge) {
38         if (isValidAge(newAge)) {
39             this.age = newAge;
40         }
41     }
42 }
43 // ❌ BAD PRACTICE
44 class BadPractice {
45     public String name;    // Anyone can modify
46     public int age;        // No validation
47     public double salary;  // Sensitive data exposed
48
49     // No encapsulation, no control
50 }
51 }
52 // ✅ GOOD PRACTICE
53 class GoodPractice {
54     private String name;
55     private int age;
56     private double salary;
57
58     // Controlled access with validation
59     public void setAge(int age) {
60         if (age >= 0 && age <= 150) {
61             this.age = age;
62         } else {
63             throw new IllegalArgumentException("Invalid age");
64         }
65     }
66 }
67
68     public int getAge() {
69         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)

```
1 class CreditCard {
2     // Private - sensitive data
3     private String cardNumber;
4     private String cvv;
5     private String pin;
6
7     // Public - safe operations
8     public boolean validatePin(String enteredPin) {
9         return this.pin.equals(enteredPin);
10    }
11
12    // Public - masked display
13    public String getMaskedCardNumber() {
14        return "****-****-****-" + cardNumber.substring(12);
15    }
16
17    // Private - internal validation
18    private boolean isValidCardNumber(String number) {
19        return number.length() == 16 && number.matches("\d+");
20    }
21 }
```

Scenario 2: Template Method Pattern

```

1 abstract class DataProcessor {
2     // Public - template method
3     public final void process() {
4         loadData();
5         validateData();
6         processData();
7         saveData();
8     }
9
10    // Protected - subclasses can override
11    protected abstract void loadData();
12    protected abstract void processData();
13
14    // Private - cannot be overridden
15    private void validateData() {
16        System.out.println("Validating data...");
17    }
18
19    // Protected - subclasses can override
20    protected void saveData() {
21        System.out.println("Saving data...");
22    }
23 }
24 class CSVProcessor extends DataProcessor {
25     @Override
26     protected void loadData() {
27         System.out.println("Loading CSV data");
28     }
29
30     @Override
31     protected void processData() {
32         System.out.println("Processing CSV data");
33     }
34 }
35 }
```

Scenario 3: Builder Pattern

```

1 class User {
2     // Private fields
3     private final String username;      // Required
4     private final String email;        // Required
5     private String firstName;        // Optional
6     private String lastName;         // Optional
7     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='" +
69             email +
70                 "', firstName='" + firstName + "', lastName='" +
71                 lastName +
72                     "', age=" + age + "}";
73     }
74
75     public class Main {
76         public static void main(String[] args) {
77             // Cannot create User directly - constructor is private
78             // User user = new User(); // ✗ Compile error
79
80             // Must use builder
81             User user = new User.UserBuilder("john_doe",
82                 "john@example.com")
83                 .firstName("John")
84                 .lastName("Doe")
85                 .age(30)
86                 .build();
87
88         }
89     }

```

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."

```
1 // No access modifier = default/package-private
2 class DefaultExample {
3     int value;    // default access
4
5     void display() { // default access
6         System.out.println("Default access");
7     }
8 }
```

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."

```

1 class Parent {
2     protected void display() {
3         System.out.println("Parent");
4     }
5 }
6 class Child extends Parent {
7     // ✅ OK – same access level
8     @Override
9     protected void display() {
10        System.out.println("Child");
11    }
12 }
13
14 // ✅ OK – increased access level
15 /*
16     @Override
17     public void display() {
18         System.out.println("Child");
19     }
20 */
21
22 // ❌ ERROR – reduced access level
23 /*
24     @Override
25     private void display() {
26         System.out.println("Child");
27     }
28 */
29 }

```

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."

```
1 class Parent {  
2     private void display() {  
3         System.out.println("Parent private");  
4     }  
5  
6     public void callDisplay() {  
7         display(); // Calls Parent's private method  
8     }  
9 }  
10 class Child extends Parent {  
11     // This is NOT overriding - it's a new method  
12     private void display() {  
13         System.out.println("Child private");  
14     }  
15  
16     public void callChildDisplay() {  
17         display(); // Calls Child's own method  
18     }  
19 }  
20 }  
21 public class Main {  
22     public static void main(String[] args) {  
23         Child child = new Child();  
24         child.callDisplay(); // Parent private  
25         child.callChildDisplay(); // Child private  
26     }  
27 }  
28 }
```

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