1. **What is a Method Signature in Java?**

Method signature means method name followed by parameter types.

In Java, a **method signature** is the **unique identity of a method**. It helps the Java compiler **differentiate** between multiple methods, especially when method overloading is used.

### ****Method Signature Includes:****

1. **Method name**
2. **Parameter list (type, number, and order)**

* The **data types** of the parameters
* The **number** of parameters
* The **order** of parameters

Together, they help Java distinguish between overloaded methods.

👉 It **does NOT include:**

* Return type
* Access modifiers (public, private, etc.)
* throws clause
* Method body

|  |
| --- |
| int add(**int** a, **int** b) add(**int, int**)  add(**int** eno, **String** ename) add(**int, string**)  add(**float** sal) add(**float**)  add() no signature |

**Why is Method Signature Important?**

* It helps Java determine **which method to call**.
* It enables **method overloading**.
* It's part of the **method declaration** but **not** affected by return type or access modifiers.

**Invalid Overloading (Same Signature, Different Return Type)**

|  |
| --- |
| public int calculate(int x) { return x; } // Not allowed:  public double calculate(int x) { return x; } // ❌ Compile-time error |

Because the **signature** is still calculate(int), and Java **doesn't allow overloading based on return type** alone.

**Note:** A class cannot contain 2 same methods with same signatures, otherwise it’s a CE.

### ****What is Method Overloading in Java?****

**Method Overloading** means having **multiple methods with the same name** in the **same class**, but with **different parameter lists** (types, number, or order).

It helps you perform **similar operations** with **different inputs**, improving **code readability and reusability**.

**Note 1:** "**Method Overloading is compile-time polymorphism, where multiple methods share the same name but differ in parameter list.**"

**Note 2:** Automatic promotion in method overloading

In method overloading if exact method match is not found then compiler doesn’t rise any CE. Compiler tries to promote the method argument to the next level and checks if any method match is available or not, if available then it will executed otherwise compiler promotes the method argument again to the next level. In this way compiler checks all the possible promotions, finally if no method is matched then compiler rises error.

### Q1: What is required for method overloading to be valid?

**A.** Different return types  
**B.** Different access modifiers  
**C.** Different number or types of parameters  
**D.** Different method names

**Q2:** Which of the following is a valid overloaded method pair?

int display(String s);

void display(String s);

**A.** Yes  
**B.** No  
(❌ Same name + same parameter list = not a valid overload, return type doesn’t count)

### ****Q3:**** Method overloading is an example of:

**A.** Runtime Polymorphism  
**B.** Compile-time Polymorphism  
**C.** Encapsulation  
**D.** Abstraction

It is resolved during compilation.

### ****Q4:**** In which of the following cases is method overloading not allowed?

**A.** void test(int a)  
int test(int a)

**B.** void test(int a)  
void test(String b)

Return type doesn't make the method signature unique.

## **Java Coding Task: Method Overloading in a Mini Calculator**

### ****Task Title:****

**"Build a Mini Calculator Using Method Overloading"**

### ****Objective:****

Practice method overloading by creating multiple versions of a method named calculate() to handle different data types and operations.

**Requirements:**

Create a class called MiniCalculator with the following overloaded calculate() methods:

1. calculate(int a, int b) → returns **sum**
2. calculate(double a, double b) → returns **product**
3. calculate(int a, int b, int c) → returns **average**
4. calculate(String operation, int a, int b)
   * if operation is "add", return sum
   * if "sub", return subtraction
   * if "mul", return multiplication
   * if "div", return division (integer division)

**Sample Code Starter (students should complete):**

|  |
| --- |
| public class MiniCalculator {  // 1. Method overloading: sum of two integers  int calculate(int a, int b) {  return a + b;  }  // 2. Method overloading: product of two doubles  double calculate(double a, double b) {  return a \* b;  }  // 3. Method overloading: average of 3 integers  int calculate(int a, int b, int c) {  return (a + b + c) / 3;  }  // 4. Method overloading: dynamic operation  int calculate(String operation, int a, int b) {  switch (operation) {  case "add": return a + b;  case "sub": return a - b;  case "mul": return a \* b;  case "div": return b != 0 ? a / b : 0;  default: return -1; // Invalid operation  }  }  public static void main(String[] args) {  MiniCalculator calc = new MiniCalculator();  System.out.println("Sum: " + calc.calculate(10, 20));  System.out.println("Product: " + calc.calculate(3.0, 4.0));  System.out.println("Average: " + calc.calculate(10, 20, 30));  System.out.println("Operation add: " + calc.calculate("add", 10, 5));  System.out.println("Operation sub: " + calc.calculate("sub", 10, 5));  }  } |

**Sample Output:**

|  |
| --- |
| **Sum:** 30  **Product:** 12.0  **Average:** 20  **Operation add:** 15  **Operation sub:** 5 |

## **What is Method Overriding in Java?**

**Method Overriding** means **redefining a method** from the **parent (super) class** in the **child (sub) class** with the **same method name, return type, and parameters**.

It allows **runtime polymorphism** — the correct version of the method is called **based on the object type** at runtime.

### Real-Life Analogy:

Think of a **base class Vehicle** with a method start().  
Different vehicles start differently — like a **Car**, **Bike**, or **Electric Scooter**.

Each subclass **overrides** the start() method to define its own way of starting.

**Key Rules of Overriding:**

|  |  |
| --- | --- |
| **Rule** | **Description** |
| Method name | Must be the same |
| Parameters | Must be the same |
| Return type | Must be the same (or covariant) |
| Access level | Cannot be more restrictive |
| Only instance methods | Static methods are not overridden |
| @Override annotation | Optional but highly recommended |

**Real-Time Example:**

|  |
| --- |
| class Bank {  double getInterestRate() {  return 5.0;  }  }  class HDFCBank extends Bank {  @Override  double getInterestRate() {  return 6.5;  }  }  class MainApp {  public static void main(String[] args) {  Bank b = new HDFCBank(); // Upcasting  System.out.println("Rate: " + b.getInterestRate()); // Output: 6.5  }  } |

**Note:** Even though reference is Bank, at **runtime**, HDFCBank method is executed — this is **runtime polymorphism**.

**Overriding is not possible for:**

* private methods
* static methods (these are **hidden**, not overridden)
* final methods (cannot be changed)
* Constructors (constructors are never inherited)

Title: Payment System Using Method Overriding

**Task Description:**

Create a base class Payment with a method processPayment().  
Create three subclasses:

* CreditCardPayment
* UPIPayment
* CashPayment

Each subclass must **override** processPayment() and print a unique message.

In the main() method, use **runtime polymorphism** to call the overridden methods.

**Starter Code:**

|  |
| --- |
| class Payment {  void processPayment() {  System.out.println("Processing payment...");  }  }  class CreditCardPayment extends Payment {  @Override  void processPayment() {  System.out.println("Processing Credit Card Payment");  }  }  class UPIPayment extends Payment {  @Override  void processPayment() {  System.out.println("Processing UPI Payment");  }  }  class CashPayment extends Payment {  @Override  void processPayment() {  System.out.println("Processing Cash Payment");  }  }  public class PaymentApp {  public static void main(String[] args) {  Payment payment;  payment = new CreditCardPayment();  payment.processPayment(); // Output: Processing Credit Card Payment  payment = new UPIPayment();  payment.processPayment(); // Output: Processing UPI Payment  payment = new CashPayment();  payment.processPayment(); // Output: Processing Cash Payment  }  } |

### ****Q1:**** Which of the following is required for method overriding?

**A.** Same method name, different parameters  
**B.** Different return type  
**C.** Same method name, same signature  
**D.** Method should be private

### ****Q2:**** Which methods cannot be overridden?

**A.** Static  
**B.** Final  
**C.** Private  
**D.** All of the above

### ****Q3:**** What type of polymorphism is method overriding?

**A.** Compile-time  
**B.** Runtime  
**C.** Static  
**D.** None

### ****Q4:**** What happens if you remove @Override annotation?

**A.** Compile error  
**B.** No runtime error but overrides silently  
**C.** Can't override  
**D.** JVM error  
(@Override is just a safeguard)

### ****Q5:**** Which of these allows method overriding?

**A.** Static methods  
**B.** Final methods  
**C.** Non-static methods  
**D.** Private methods

### ****Q6:**** Can a constructor be overridden?

**A.** Yes  
**B.** No  
Explanation: Constructors are not inherited.

### ****Q7:**** What keyword is used to ensure a method is overridden correctly?

**A.** @Static  
**B.** override()  
**C.** @Override  
**D.** this.override

### ****Q8:**** What is runtime polymorphism?

**A.** Multiple methods with same name in one class  
**B.** Method resolution at compile time  
**C.** Method resolution at runtime  
**D**. None of the above

### ****Q9:**** Can the return type differ while overriding?

**A.** No  
**B.** Only if covariant  
**C.** Yes, any type  
**D.** Only if access modifier is same

## **Overloading vs Overriding Summary Table**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Overloading** | **Overriding** |
| Class type | Same class | Different class (Inheritance) |
| Parameters | Must differ | Must be the same |
| Return type | Can differ | Must be same or covariant |
| Runtime/Compile time | Compile-time (early binding) | Runtime (late binding) |
| Access to superclass | Not required | Required (superclass needed) |