Chapter 6: Methods

6.1. Introduction

Definition

- **Method**: A block of code that performs a specific task.
- **Belongs to a Class**: In object-oriented programming, methods are functions tied to a class.
- **Purpose**: Defines what an object can do (its behavior).
- **How to Use**: Call the method to execute its code and perform the task.

Why Use Methods?

- **Problem**: Repeating code is inefficient and hard to maintain.
- **Solution**:
  - **Reuse Code**: Write once, use anywhere.
  - **Simplify**: Make code easier to read and debug.
  - **Modularize**: Break problems into smaller, manageable parts.

Challenge

Example: Finding the sum of ranges (e.g., 1-10, 20-37) using repetitive code

```
int sum1 = 0;
for (int i = 1; i <= 10; i++) {
    sum1 += i;
}

int sum2 = 0;
for (int i = 20; i <= 37; i++) {
    sum2 += i;
}
```

Drawbacks: Repetitive code, hard to maintain.

Solution

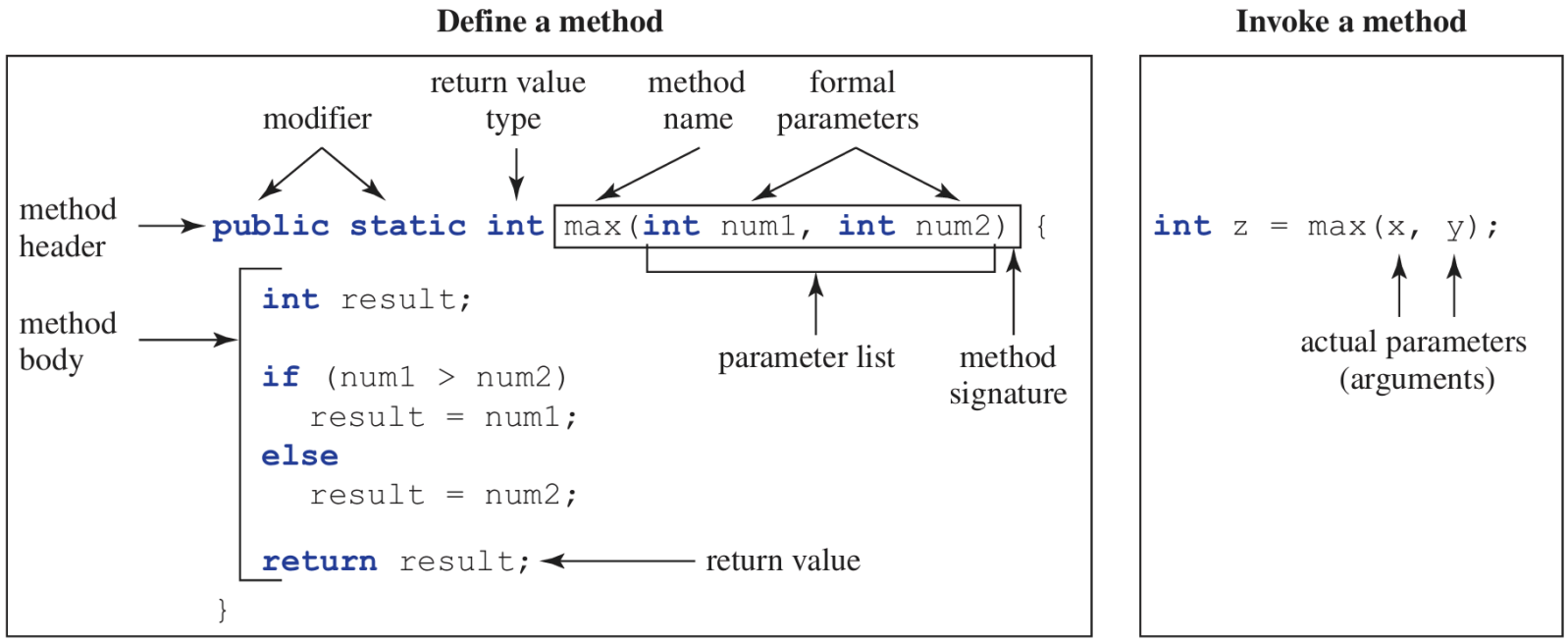
Example: Finding the sum of ranges (e.g., 1-10, 20-37) using method

```
public static int sum(int i1, int i2) {
    int result = 0;
    for (int i = i1; i <= i2; i++) {
        result += i;
    }
    return result;
}
```

Advantage: Flexible use: `sum(1, 10)`, `sum(20, 37)`, etc.

6.2. Defining a Method

Anatomy of a Method



Syntax of a Method

Syntax: Method Declaration

```
modifier returnValueType methodName(list of parameters) {
    // Method body
}
```

Syntax: Method Invocation

```
methodName(argument1, argument2, ...);
```

Components of a Method:

- **Modifier**: Defines who can access the method (e.g., `public`, `private`).
- **Return Type**: The type of value the method gives back (e.g., `int`, `void`).
- **Method Name**: The name used to call the method (e.g., `max`).
- **Parameters**: Inputs the method needs to work (e.g., `int num1, int num2`).
- **Body**: The code inside the method that does the work.
- **Return Statement**: Sends the result back to the caller.

Type of Methods Modifiers

Modifier	What It Does?
default	Adds a default method in interfaces.
public	Makes the method accessible from any class.
private	Restricts access to within the same class only.
protected	Allows access within the same package and subclasses.
static	Links the method to the class, not an object.
final	Prevents the method from being changed or overridden.
abstract	Declares a method without implementation; must be defined in a subclass.

Modifier	What It Does?
synchronized	Ensures only one thread can use the method at a time (thread-safe).
strictfp	Makes floating-point calculations consistent across platforms.
native	Indicates the method is written in another language like C.
transient	Excludes the variable from being saved during serialization.
volatile	Ensures the variable is always read from main memory, not a cached copy.

Example: Define a method to find the maximum of two numbers.

```
public static int max(int num1, int num2) {
    if (num1 > num2)
        return num1;
    else
        return num2;
}
```

Example: Invoke the `max` method

```
int larger = max(3, 4);
System.out.println(max(3, 4));
```

6.3. Calling a Method

Types of Method Calls

Example: Call the value-returning method.

```
int larger = max(3, 4);
System.out.println(max(3, 4));
```

Example: Call the void method.

```
printMessage("Hello, World!");
```

Flow of Control

- **When Called**: Program control is transferred to the method.
- **When Completed**: Control returns to the caller after execution.

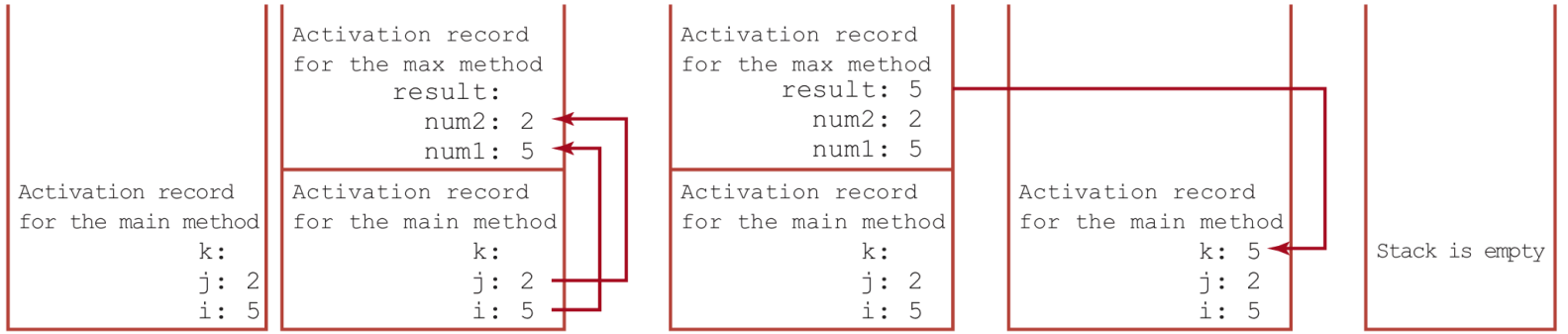
Example: Maximum of Two Numbers

```
public static void main(String[] args) {
    int i = 5, j = 2;
    int k = max(i, j);
    System.out.println("The maximum of " + i + " and "
        + j + " is " + k);
}

public static int max(int num1, int num2) {
    return (num1 > num2) ? num1 : num2;
}
```

Explanation:

- The `max` method is invoked from `main`.
- Values `i` and `j` are passed to `num1` and `num2`.
- The result is returned and stored in `k`.



(1) The `main` method is invoked.

(2) The `max` method is invoked.

(3) The `max` method is being executed.

(4) The `max` method is finished and the return value is sent to `k`.

(5) The `main` method is finished.

Key Notes

- **Return Statement**: Required for value-returning methods.
- **Void Methods**: Do not require a return statement but can use `return;` to terminate early.
- **Reusable Code**: Methods make coding easier by allowing you to write code once and use it multiple times.

6.4. Void vs. Value-Returning Methods

What is Void Method?

- **Definition**: A void method does not return any value.
- **Purpose**: Perform an operation or action.
- **Invocation**: Must be called as a statement.

Example: Define a void method

```
public static void printMessage(String message) {
    System.out.println(message);
}
```

Example: Call a void method

```
printMessage("Hello, World!");
```

What Are Value-Returning Methods?

- **Definition**: A method that returns a value to the caller.
- **Purpose**: Perform an operation and provide a result.
- **Invocation**: Can be used in expressions or assigned to variables.

Example: Define a value-returning method

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

Example: Call a value-returning method

```
int sum = add(5, 10);
System.out.println(sum);
```

Key Differences

Feature	Void Method	Value-Returning Method
Return Type	`void`	Data type (e.g., `int`)
Return Statement	Optional (`return;`)	Required return with a value
Usage	Standalone statement	Assign to variables or expressions

Comparison Example

Example: Printing vs. Returning Grades (Void Method)

```
public static void printGrade(double score) {
    if (score >= 90.0) System.out.println('A');
    else if (score >= 80.0) System.out.println('B');
    else if (score >= 70.0) System.out.println('C');
    else System.out.println('F');
}
```

Example: Returning Grades (Value-Returning Method)

```
public static char getGrade(double score) {
    if (score >= 90.0) return 'A';
    else if (score >= 80.0) return 'B';
    else if (score >= 70.0) return 'C';
    else return 'F';
}
```

When to Use Void vs. Value-Returning Methods

- **Void Methods**: When no value needs to be returned (e.g., logging, printing).
- **Value-Returning Methods**: When a result is needed for further operations.

6.5. Passing Arguments by Values

What is Passing Arguments by Values?

- **Definition**: When a method is invoked, the value of the argument is passed to the method's parameter.
- **Key Point**:
  - The method works with the value passed, not the actual variable.
  - Changes made to the parameter inside the method do not affect the argument outside.

Example: Incrementing a value inside a method.

```
public class Increment {
    public static void main(String[] args) {
        int x = 1;
        System.out.println("Before the call, x is " + x);
        increment(x);
        System.out.println("After the call, x is " + x);
    }

    public static void increment(int n) {
        n++;
        System.out.println("n inside the method is " + n);
    }
}
```

Output:

```
Before the call, x is 1
n inside the method is 2
After the call, x is 1
```

Explanation:

- The value of `x` remains unchanged after the method call.
- The method increments the value of `n` inside the method.
- The original value of `x` is not affected.
- The method works with a copy of the value of `x`.
- The method does not have access to the original variable `x`.
- The method only modifies the copy of the value.

Example: Swapping two values inside a method.

```
public static void swap(int n1, int n2) {
    int temp = n1;
    n1 = n2;
    n2 = temp;
}
```

Explanation:

- Inside the method, the values of `n1` and `n2` swap.
- The original arguments (outside the method) remain unchanged.

Key Notes on Pass-By-Value

- **Matching Parameters with Arguments**:
  - Arguments must align with parameters in:
    - **Order**: Follow the sequence of parameters.
    - **Number**: Match the exact count of parameters.
    - **Type**: Use compatible data types.
- **Key Effects**:
  - The original variables outside the method remain unchanged.
  - This protects the original data from being accidentally modified.

When and Why to Use Pass-By-Value?

- **Advantages**:
  - Keeps the original data safe from changes.
  - Makes methods work independently without side effects.
- **Best Use Cases**:
  - When you need to perform calculations or tasks without altering the original data.

Tip: Pass an Array as an Argument

Example: Summing a variable number of arguments.

```
public static void main(String[] args) {
    int[] numbers = {1, 2, 3, 4, 5};
    System.out.println(sum(numbers));
}

public static int sum(int... numbers) {
    int total = 0;
    for (int number : numbers) {
        total += number;
    }
    return total;
}
```

Explanation:

- The `...` syntax allows you to pass a variable number of arguments.

6.6. Modularizing Code

What is Modularizing Code?

- **What is Modularizing Code?** Breaking code into smaller, easy-to-manage methods.
- **Why is it Useful?**
  - **Easier to Read**: Makes the logic simple to follow.
  - **Easier to Fix**: Errors are limited to specific methods.
  - **Reusable**: You can use the same methods in other programs.

Example: Calculate the greatest common divisor (GCD) of two numbers. [តួចែករួមធំបំផុត]

```
public static int gcd(int n1, int n2) {
    int gcd = 1;
    int k = 2; // Potential gcd
    while (k <= n1 && k <= n2) {
        if (n1 % k == 0 && n2 % k == 0)
            gcd = k; // Update gcd
        k++;
    }
    return gcd;
}
```

Explanation:

- The GCD logic is written as a separate method.
- This makes the code reusable in other programs.
- It simplifies the main program by focusing on the GCD calculation in one place.
- Easier to debug and understand.
- Promotes modular programming.

Advantages of Modular Design

- **Avoids Repetition**: No need to write the same code multiple times.
- **Keeps Code Organized**: Each method solves one specific problem.
- **Easy to Update**: Fix or change one method without affecting others.

Applying Modular Design

- **Breaking Down Complex Problems Using Methods**:
  - **Example**: Prime Number [ចំនួនបឋម] Program
    - **Submethods**:
      - `isPrime`: Checks if a number is prime.
      - `printPrimeNumbers`: Prints a list of prime numbers.
  - **Benefits**:
    - **Clearer Logic**: Each method focuses on one task.
    - **Reusable Code**: Methods can be used in other programs.

6.7. Case Study: Converting Hexadecimals to Decimals

Practice.

6.8. Overloading Methods

What is Method Overloading?

- **What is Method Overloading?** Method overloading happens when multiple methods have the **same name** but differ in:
  - **Number of parameters** (e.g., one method takes 2 inputs, another takes 3).
  - **Type of parameters** (e.g., one method uses `int`, another uses `double`).
  - **Order of parameters** (e.g., `int, double` vs. `double, int`).

- **Why Use Method Overloading?**
  - Makes your code easier to read and understand.
  - Lets you use the same method name for similar tasks, making it more flexible.

Example of Overloading

Example: Overloading the `max` method to handle different data types.

```
public static int max(int num1, int num2) {
    return (num1 > num2) ? num1 : num2;
}

public static double max(double num1, double num2) {
    return (num1 > num2) ? num1 : num2;
}

public static double max(double num1, double num2, double num3) {
    return Math.max(num1, Math.max(num2, num3));
}
```

Explanation:

- The `max` method is designed to work with different types of numbers.
- One version works with two integers.
- Another version works with two decimal numbers (doubles).
- A third version can handle three decimal numbers.

Benefits of Overloading Methods

- **Better Code Reuse**: Use the same method name for similar tasks, no need to create many different names.
- **Easier to Maintain**: Group related tasks under one method name for better organization.
- **Simpler to Use**: Makes the code easier to understand and work with for everyone.

Rules for Overloading

- Methods can have the **same name** but must differ in:
  - **Number of parameters** (e.g., 2 inputs vs. 3 inputs).
  - **Type of parameters** (e.g., `int` vs. `double`).
  - **Order of parameters** (e.g., `int, double` vs. `double, int`).
- **Important**: The return type **cannot** be the only difference between overloaded methods.

Ambiguous Overloading

- **What is Ambiguity?** When the compiler cannot determine which method to call due to similar parameter types.

Example: Ambiguous Overloading

```
public static void print(int num, double value) { ... }
public static void print(double value, int num) { ... }
```

Explain:

- Ambiguity arises if you call `print(5, 5);`.
- Compiler cannot determine which method to invoke.
- Solution: Avoid ambiguous overloading.

6.9. The Scope of Variables

What is the Scope of a Variable?

- **What is Variable Scope?** The scope of a variable is the part of the program where you can use the variable.

Local Variables

- **What Are Local Variables?**
  - Variables declared inside a method, block, or constructor.
  - They exist only while the method or block is running.
  - You cannot use them outside the method or block where they are defined.

Example:

```
public void calculateSum() {
    int sum = 0; // Local variable
    for (int i = 1; i <= 10; i++) {
            sum += i; // Updating the local variable
    }
    System.out.println(sum); // Can use 'sum' only inside this method
}
```

Key Point: Local variables are temporary and only used where they are created.

Instance Variables

- **What Are Instance Variables?**
  - Declared inside a class but outside any methods.
  - Each object gets its own copy of these variables.

Example:

```
public class Student {
    String name; // Instance variable
    int age;     // Instance variable

    public void displayInfo() {
            System.out.println("Name: " + name);
            System.out.println("Age: " + age);
    }
}
```

Key Points:

- Instance variables store data specific to each object.
- You can access them using the object of the class.
- Example usage: `student1.name = "Alice";`
- Makes it easy to manage object-specific information.

Class Variables

- **What Are Class Variables?**
  - Declared with the `static` keyword.
  - Shared by all objects of the class.
  - Only one copy of the variable exists for all objects.

Example:

```
public class Employee {
    static int employeeCount; // Shared by all employees

    public static void increaseCount() {
            employeeCount++; // Increment the shared count
    }
}
```

Explanation:

- The `employeeCount` variable is shared by all `Employee` objects.
- You can access it using the class name: `Employee.employeeCount`.
- Useful for storing information that is common to all objects.

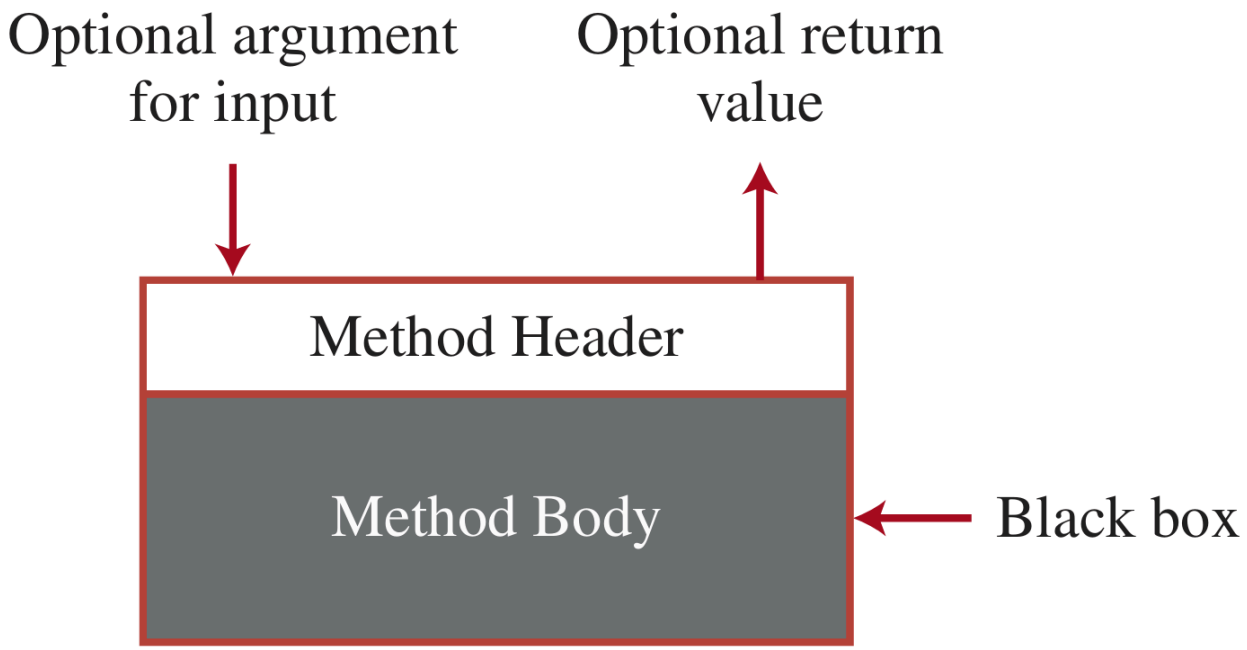
Key Notes on Variable Scope

- **Block Scope**: Variables declared inside `{}` can only be used within those braces.
- **Method Scope**: Variables declared inside a method can only be used in that method.
- **Class Scope**: Instance variables and static variables can be used anywhere in the class.
- **Lifetime**:
  - **Local Variables**: Exist only while the method or block is running.
  - **Instance Variables**: Exist as long as the object exists.
  - **Static Variables**: Exist as long as the program runs.

6.10. Case Study: Generating Random Characters

Practice.

6.11. Method Abstraction and Stepwise Refinement


Optional argument for input
Optional return value
Method Header
Method Body
Black box

What is Method Abstraction?

- **What is Method Abstraction?** Focuses on **what** a method does, not **how** it works. Hides the details of how the method is implemented.
- **Why Use It?** Makes code easier to read and understand. Encourages breaking problems into smaller, reusable parts.
- **Benefits**:
  - **Simplifies Code**: Focus on the big picture, not the details.
  - **Reusable**: Use the same method in different programs.
  - **Scalable**: Easy to add new features without changing existing code.

Stepwise Refinement Process

1. **Break the Problem into Smaller Parts**: Divide the big problem into smaller, easier-to-solve pieces.
2. **Solve Each Part Step by Step**: Focus on solving one small part at a time.
3. **Put Everything Together**: Combine the solutions of all the small parts to solve the entire problem.

Example: Printing Prime Number [ចំនួនបឋម]

- **Step 1**: Write `isPrime` method to check if a number is prime.
- **Step 2**: Write `printPrimeNumbers` method to print prime numbers.
- **Step 3**: Combine methods to create the final solution.

Example: Printing Prime Numbers

```
public static boolean isPrime(int number) {
    for (int divisor = 2; divisor <= number / 2; divisor++) {
        if (number % divisor == 0)
            return false;
    }
    return true;
}

public static void printPrimeNumbers(int count) {
    int number = 2;
    int printed = 0;
    while (printed < count) {
        if (isPrime(number)) {
            System.out.println(number);
            printed++;
        }
        number++;
    }
}
```

Concepts of Designing Abstract Methods for Project

Example: Project of making a Calculator App.

- Methods for input validation: `validateInput`.
- Methods for mathematical operations: `add`, `subtract`, `multiply`, `divide`, `mod`, `power`, `sqrt`, `abs`, `round`, `ceil`, `floor`, `min`, `max`, `average`, `factorial`, `sin`, `cos`, `tan`, `log`, `ln`, `log_x`, `getGCD`, `getLCM`, `convertToBinary`, `convertToHex`, `convertToOctal`, `convertToDecimal`.
- Methods for input management: `delete`, `clear`.
- Methods for data persistence: `save`, `load`.
- ...

Example: Project login and registration App.

- Methods for user input: `getUsername`, `getPassword`.
- Methods for validation: `checkUsername`, `checkPassword`, `validatePassword`, `validateEmail`, `validatePhone`.
- Account operations: `login`, `logout`, `register`, `deleteAccount`, `changePassword`, `forgotPassword`.
- Data management: `save`, `load`.
- ...

Example: Project of making a Library Management System.

- Methods for book management: `addBook`, `removeBook`, `searchBook`, `validateBook`.
- Methods for member management: `addMember`, `removeMember`, `searchMember`, `validateMember`.
- Methods for librarian management: `addLibrarian`, `removeLibrarian`, `searchLibrarian`, `validateLibrarian`.
- Methods for transactions: `borrowBook`, `returnBook`.
- Methods for stock management: `updateStock`, `checkStock`, `getStockHistory`.
- ...

Example: Project of making a Banking System.

- Methods for account operations: `deposit`, `withdraw`, `transfer`, `checkBalance`.
- Methods for account management: `addAccount`, `removeAccount`, `searchAccount`, `validateAccount`.
- Methods for transaction validation: `validateTransaction`.
- Methods for transaction history: `getTransactionHistory`.
- Methods for interest calculation: `calculateInterest`.
- Methods for loan management: `applyLoan`, `approveLoan`, `rejectLoan`.
- ...

End of the Chapter