In the previous chapter, we extended our control flow capabilities by implementing **loops** (such as `while`, `do-while`, and `for`) to execute blocks of code repeatedly and by introducing **File I/O** for saving and retrieving data persistently.

Now we will transition from controlling *when* statements execute to mastering *how* to organize them efficiently through **methods**.

What are Methods?

Methods are **reusable collections of statements** that perform specific tasks, enabling essential programming principles:

- **Functional decomposition** (breaking large problems into smaller, manageable pieces)
- **Code reuse**

What you will learn

In this chapter, you will learn how to: - Understand the role of methods in functional decomposition and code reuse - Distinguish between **void** methods and **value-returning** methods - Define a method using a method header (signature) and a body - Understand the role of modifiers, particularly **static** and **instance** methods - Pass arguments (values) to a method's parameters - Understand the difference between **passing by value** (for primitives) and **passing object references** - Manage **local variables**, including their scope and lifetime - Return values and object references from methods - Use methods for effective problem solving (functional decomposition) # 5.1 Introduction to Methods

Methods are foundational components in Java programming. They are reusable pieces of code, sometimes called **functions** in other languages, that allow you to perform a specific task, often on an object.

The use of methods enables two primary benefits:

- **Code reuse**
- **Functional decomposition**—the process of breaking a large, complex problem down into smaller, manageable sub-problems

Method Structure

Every method consists of two parts:

1. **Header** (or signature)
2. **Body**

Method Header

The **Method Header** defines the method's characteristics and includes:

1. **Method Modifiers** (e.g., `public`, `static`)
2. **Return Type** (e.g., `void`, `int`, `String`)
3. **Method Name**
4. **Parameter List** (written in parentheses)

Here are some examples of method headers:

```
public static void printName(String name)
/*
Method Name: printName
Modifiers: public, static
Return type: nothing
Input Parameters: name (String)
*/

public static int squareNumber(int number)
/*
Method Name: squareNumber
Modifiers: public, static
Return type: int
Input Parameters: number (int)
*/

public static String greetStudent(String studentName)
/*
Method Name: greetStudent
Modifiers: public, static
Return type: String
Input Parameters: studentName (String)
*/

public static String greetStudentFromTeacher(String studentName, String teacherName)
/*
Method Name: greetStudentFromTeacher
Modifiers: public, static
Return type: String
Input Parameters: studentName (String), teacherName (String)
*/
```

Method Body

The **Method Body** contains the collection of statements enclosed in curly braces `{}` that are executed when the method is called.

```
public static void greetUser(String name) {
    // Method body starts here with opening brace
    System.out.println("Welcome, " + name + "!");
```

```
    System.out.println("Hope you enjoy learning Java.");
    // Method body ends here with closing brace
}
```

Method Types: Void vs. Value-Returning

Methods are typically categorized by what they return:

1. **Void Methods:** These methods perform a task and then terminate without sending any value back to the calling code. They are often used for tasks like displaying information (e.g., printing to the console).
2. **Value-Returning Methods:** These methods perform a task (often a calculation or retrieval) and use the `return` statement to send a single value or object reference back to the code that called it. The return type must be explicitly defined in the method header.

Method Modifiers: Static vs. Instance

Method modifiers define the access and behavior of a method. The most important non-access modifier to understand is `static`.

- **Static Methods** (Class Methods): These methods **belong to the class itself** rather than any specific instance (object). They can be called directly on the class name without needing to create an object first, such as `Math.random()`.
- **Instance Methods** (Non-Static Methods): These methods **belong to an object** and operate on that object's specific data (fields). They require you to create an instance of the class (using `new`) before they can be called, such as `myRandom.nextInt(100)`.

Before we introduce writing our own classes, our methods will primarily be **static**.

Calling a Method

A method only executes when it is called. The `main` method is special because the Java Virtual Machine (JVM) calls it automatically when the program starts. Other methods must be executed by a **method call statement**.

A method call statement includes:

- The method name
- Arguments (if any) inside parentheses
- Followed by a semicolon

Importantly, method modifiers and the return type are **not** included in the method call statement; they are only used in the method header.

Example: Defining and Calling a Simple Void Method

This example demonstrates a simple `static` method that is called from `main`.

```
public class SimpleMethod {
   public static void displayHeader() { // Method definition (Header + Body)
      // Method Body
      System.out.println("----------------------------------------");
      System.out.println("  Method displayHeader() is Executing   ");
      System.out.println("----------------------------------------");
   }

   public static void main(String[] args) {
      System.out.println("Program starting...");
      displayHeader(); // Method Call
      System.out.println("Program ending.");
   }
}
```

Layered Method Calls

Methods can call other methods, leading to **layered method calls** (or a call stack). When a called method finishes execution, control is returned to the line immediately following the call in the calling method.

When method A calls B, B is called and finishes, then control returns to A.

```
public class LayeredCallDemo {
    public static void methodA() {
        System.out.println("  2.  In methodA: Calling methodB.");
        methodB(); // Execution pauses here and jumps to methodB
        System.out.println("  4.  In methodA: Back from methodB. Finishing methodA.");
    }

    public static void methodB() {
        System.out.println("      3.  In methodB: Deepest point.");
        // methodB finishes, control returns to methodA
    }

    public static void main(String[] args) {
        System.out.println("1.  In main: Calling methodA.");
        methodA(); // Program execution pauses here and jumps to methodA
        System.out.println("5.  In main: Back from methodA. Program finishing.");
    }
}
```

Here's the output of the `LayeredCallDemo` program:

```
1.  In main: Calling methodA.
```

```
  2.  In methodA: Calling methodB.
    3.  In methodB: Deepest Point.
  4.  In methodA: Back from methodB. Finishing methodA.
5.  In main: Back from methodA. Program finishing.
```

Check Your Understanding (5.1)

1. What is the primary purpose of breaking a program down into smaller pieces (methods)?
2. Name the two main parts of a method declaration.
3. What is the return type of a method that performs a task but does not send any value back to the calling code?
4. Explain the difference between a `static` method and an instance method, and give an example of a built-in Java static method.

Practice Problems (5.1)

1. Write a `public static void` method named `printGreeting` that takes no parameters and prints "Welcome to the Java Textbook Program!" to the console. Show how you would call this method from the `main` method.
2. Given the following method definition, write the single correct statement that would call this method: `public static void calculateArea(double length, double width)` # 5.2 Passing Arguments to a Method

Methods become flexible and reusable when you allow them to accept **arguments** (or input).

Parameters vs. Arguments

It is crucial to distinguish between the terminology:

- **Parameter:** The variable declared in the method header (definition) that acts as a placeholder to accept the value being passed in.
- **Argument:** The actual value, variable, or literal that is passed to the method when it is called.

The values you pass as arguments are received by the parameter variables in the method body in the order they appear.

Example: Method with a Single Parameter

If a method takes one input parameter, its functionality can change based on the argument provided.

```java
public static void welcomeUser(String name) { // 'name' is the parameter
   System.out.println("Hello, " + name + "!");
}
```

```java
public static void main(String[] args) {
   welcomeUser("Alice"); // "Alice" is the argument
   welcomeUser("Bob");   // "Bob" is the argument
}
```

Arguments and Parameter Data Type Compatibility

When calling a method, the data type of the argument must be compatible with the data type of the corresponding parameter variable.

Java typically allows **widening conversions** automatically (e.g., passing an `int` argument to a `double` parameter), but strict adherence to the parameter types is necessary.

If a method expects a `double` and then an `int`, you must provide a `double` followed by an `int`.

Passing by Value (Primitives)

In Java, all primitive arguments (like `int`, `double`, `boolean`, `char`) are **passed by value**. This means that when you pass a primitive variable to a method, **only a copy of its value is passed** to the parameter variable.

Changes made to the parameter inside the method do not affect the original argument outside the method.

Example: Pass by Value

```java
public class PassByValue {
   public static void changeValue(int num) {
      num = 100; // Changes the COPY stored in parameter 'num'
      System.out.println("Inside method: num = " + num);
   }

   public static void main(String[] args) {
      int x = 10;
      System.out.println("Before call: x = " + x); // x is 10
      changeValue(x); // A copy of 10 is passed
      System.out.println("After call: x = " + x);  // x is still 10
   }
}

/*
Output:
Before call: x = 10
Inside method: num = 100
After call: x = 10
*/
```

Passing Object References

When an object (a non-primitive type, like `String`, `Scanner`, or a custom class) is passed as an argument, it is actually a **copy of the reference (memory address)** to the object that is passed.

Because the method receives a reference to the actual object in memory, the method can access and modify the original object's fields and call its methods.

The one major exception is the **String class**, which is **immutable**—meaning its contents cannot be changed once the object is created. Operations that appear to change a String actually create a new String object.

Example: Passing a Custom Object Reference

If you pass a custom object (like a `Car` object), the receiving method can update the object's instance fields.

```
// Assume Car class has a public setter method: setAvailable(boolean status)
public class ObjectPasser {
    // Method that accepts a Car object reference
    public static void updateCarStatus(Car carReference) {
        // Use the reference to modify the original object in memory
        carReference.setAvailable(false);
        System.out.println("Car status updated inside method.");
    }

    public static void main(String[] args) {
        // Car myCar = new Car(true); // Assume initial status is true
        // updateCarStatus(myCar);
        // // After the call, myCar's actual status has been permanently changed to false
    }
}
```

Check Your Understanding (5.2)

1. What is the fundamental difference between a parameter and an argument?
2. If you pass an `int` variable (a primitive) to a method, and the method internally changes the value of the parameter, why does the original `int` variable remain unchanged?
3. When a method receives an object as an argument, what specific piece of information is actually passed to the parameter variable?
4. Given the method header `public static void processData(int age, double salary)`, which of the following method calls is valid: `processData(30.0, 50000)` or `processData(30, 50000.0)`? Explain why.

Practice Problems (5.2)

1. Write a static void method called `calculateTotal` that accepts two parameters: a `double` named `price` and an `int` named `quantity`. The method should calculate and print the total cost (`price * quantity`). Show how to call this method passing a price of `15.99` and a quantity of `5`.
2. Assume you have a custom object `Student`. Write a method header for a method named `displayStudentInfo` that accepts one argument of type `Student`. Explain why this method would be typically declared as `static` if it resides in an executable class. # 5.3 More About Local Variables

A **local variable** is a variable declared inside the body of a method. Parameter variables themselves also behave similarly to local variables.

Scope

A variable's **scope** is the part of the program where that variable is accessible by its name. Local variables and parameter variables have very limited scope: **they are only accessible within the method in which they are declared**.

This limited scope means that if you have two different methods, they can each declare a local variable with the exact same name, and there will be no conflict because the methods cannot see each other's local variables.

Lifetime

The **lifetime** of a local variable is the duration for which it exists in memory. A method's local variables exist only while that method is actively executing. When the method returns control to the caller, its local variables and parameter variables are destroyed, and any values they stored are lost.

Initialization

Unlike class fields (which are automatically initialized to default values like 0 or `null`), **local variables are not automatically initialized**. They must be explicitly given a value before they can be used in an expression; otherwise, a compiler error will occur.

Example: Local Variable Scope and Lifetime

This example demonstrates that variables in one method cannot be accessed by another, even if they share the same name.

```
public class LocalVars {
   public static void methodA() {
      int count = 1; // Local variable 'count' in methodA
      System.out.println("Method A Count: " + count);
```

```
    }

    public static void methodB() {
        int count = 50; // Local variable 'count' in methodB (a different variable)
        System.out.println("Method B Count: " + count);
    }

    public static void main(String[] args) {
        methodA();
        methodB();

        // The local 'count' variables from methodA and methodB
        // are inaccessible here and are destroyed when their respective methods finish.
    }
}
/*
Output:
Method A Count: 1
Method B Count: 50
*/
```

Lifetime

The **lifetime** of a local variable is the duration for which it exists in memory. A method's local variables exist only while that method is actively executing. When the method returns control to the caller, its local variables and parameter variables are destroyed, and any values they stored are lost.

Initialization

Unlike class fields (which are automatically initialized to default values like 0 or `null`), **local variables are not automatically initialized**. They must be explicitly given a value before they can be used in an expression; otherwise, a compiler error will occur.

Example: Local Variable Scope and Lifetime This example demonstrates that variables in one method cannot be accessed by another, even if they share the same name.

```
public class LocalVars
{
    public static void methodA()
    {
        int count = 1; // Local variable 'count' in methodA
        System.out.println("Method A Count: " + count);
    }
```

```
    public static void methodB()
    {
        int count = 50; // Local variable 'count' in methodB (a different variable)
        System.out.println("Method B Count: " + count);
    }

    public static void main(String[] args)
    {
        methodA();
        methodB();

        // The local 'count' variables from methodA and methodB
        // are inaccessible here and are destroyed when their respective methods finish.
    }
}
/*
Output:
Method A Count: 1
Method B Count: 50
*/
```

Check Your Understanding (5.3)

1. What determines the scope of a local variable declared inside a method?
2. If two different methods in the same class both declare an `int` variable named `i`, will this cause a naming conflict or compiler error? Why or why not?
3. What is the lifetime of a local variable?
4. Do local variables need to be initialized before they are used?

Practice Problems (5.3)

1. Write two methods, `firstMethod` and `secondMethod`. In `firstMethod`, declare a local variable `x` and initialize it to 10. In `secondMethod`, try to print the value of `x`. Write a brief comment explaining why this will result in a compiler error.
2. Given the following method header: `public static void process(double rate)`, what is the scope of the `rate` parameter variable? # 5.4 Returning a Value from a Method

Value-returning methods execute a task and then use the `return` keyword to send a result back to the calling statement.

Defining a Value-Returning Method

To define a value-returning method, you must replace the `void` keyword in the method header with the data type of the value you intend to return (e.g., `int`,

`double`, `String`, or a custom object). A method can only have **one return type**.

The method body must contain a `return` statement that specifies the value to be sent back. This return value must be compatible with the return type defined in the header.

Example: Calculating and Returning a Value

This method calculates the area of a rectangle and returns the result as a `double`.

```
public static double calculateArea(double length, double width) {
    double area = length * width;
    return area; // Sends the calculated value back to the caller
}

public static void main(String[] args) {
    double len = 10.0;
    double wid = 5.0;

    double result = calculateArea(len, wid);
    // The result of calculateArea is assigned to the 'result' variable

    System.out.printf("The area is: %.2f%n", result);
}
```

Returning a Boolean Value

Methods can return a `boolean` value (`true` or `false`). This is often used to check conditions or status within the program logic.

Example: Returning a Boolean

The `isPassing` method returns true if the score is greater than or equal to 60.0, otherwise false:

```
public static boolean isPassing(double score) {
    return score >= 60.0;
}

public static void main(String[] args) {
    double myScore = 75.5;
    if (isPassing(myScore)) {
        System.out.println("Congratulations, you passed!");
    }
}
```

This example calls the method and uses the returned boolean value directly.

Returning Objects from Methods

Methods are not limited to returning primitives; they can return references to objects, including standard Java API objects (like a `String[]` array or `File`) or custom objects. When returning an object, a copy of the object's memory address is returned, not a copy of the object itself.

Example: Returning a File Object

The return type indicates a `File` object is being returned.

```
import java.io.File;

public class FileReturner {
    // Method returns a File object
    public static File getConfigFile() {
        File configFile = new File("config.txt");
        return configFile; // Returns a reference to the File object
    }

    public static void main(String[] args) {
        // The returned File reference is assigned to a new File variable
        File myConfig = getConfigFile();

        // Check if the file exists
        if (myConfig.exists()) {
            System.out.println("Config file found: " + myConfig.getName());
        } else {
            System.out.println("Config file not found.");
        }
    }
}
```

Check Your Understanding (5.4)

1. What must replace the `void` keyword in the method header when defining a value-returning method?
2. What keyword is mandatory inside the body of a value-returning method to send the result back to the caller?
3. When a method returns a custom object (e.g., a `Car`), is a copy of the object returned, or is something else returned?
4. Is the following method header valid? Why or why not? `public static int calculate(int x, int y)` if the method body returns `(double) x / y`.

Practice Problems (5.4)

1. Write a static method called `getSum` that accepts two `int` arguments and returns their sum as an `int`. In your main method, call `getSum` and print the result.
2. Write a method header for a method named `generateIDs` that returns an array of `long` data types. # 5.5 Problem Solving with Methods

The primary goal of methods is to simplify complex programs.

Functional Decomposition

The process of breaking a large, complex problem down into smaller, more manageable sub-problems is called **functional decomposition** or sometimes the "divide and conquer" method.

Each sub-problem is solved by a separate method, making the overall program easier to design, code, debug, and maintain. The `main` method often serves as the orchestrator, calling the various sub-methods in the correct sequence to execute the complete program logic.

Example: Functional Decomposition

```
import java.util.Scanner;

public class StudentInfo {
    public static void main(String[] args) {
        String studentName = getStudentName();
        double score = getStudentScore(studentName);
        char grade = calculateGrade(score);
        displayGradeReport(studentName, score, grade);
    }

    public static String getStudentName(){
        Scanner kbd = new Scanner(System.in);
        System.out.println("Type in your name: ");
        String name = kbd.nextLine();
        kbd.close();
        return name;
    }

    public static double getStudentScore(String studentName){
        double score;
        if (studentName.equalsIgnoreCase("Jackson")){
            score = 88;
        }
        else if (studentName.equalsIgnoreCase("Molly")){
            score = 96;
```

```
        }
        else if (studentName.equalsIgnoreCase("Peter")){
            score = 52;
        }
        else {
            score = 82;
        }
        return score;
    }

    public static char calculateGrade(double score){
        char grade;
        if (score > 90){
            grade = 'A';
        }
        else if (score > 80){
            grade = 'B';
        }
        else if (score > 70){
            grade = 'C';
        }
        else if (score > 60){
            grade = 'D';
        }
        else {
            grade = 'F';
        }
        return grade;
    }

    public static void displayGradeReport(String studentName, double score, char grade){
        System.out.println("Student name: " + studentName);
        System.out.println("Student score: " + score);
        System.out.println("Student grade: " + grade);
    }
}
```

Calling Methods That Throw Exceptions

When a method performs an operation that might fail (such as file input/output), Java requires that the method either handle the unexpected event (an **exception**) or declare that it throws the exception.

If Method A calls Method B, and Method B is defined to `throw` an exception (like `FileNotFoundException` or `IOException`) because it interacts with the file system, Method A must also declare that it `throws` the same exception or

handle it using a `try-catch` block.

Example 5.5.2: Exception Propagation

```
// Method that interacts with external resources (files), requiring throws IOException
public static void readFileData() throws IOException {
    // Code here uses Scanner/File/PrintWriter, which might throw an IOException
}

public static void main(String[] args) throws IOException {
    // Main must also declare 'throws IOException' or handle it,
    // since it calls readFileData().
    readFileData(); // This call forces main to declare the exception
    System.out.println("Data processed successfully.");
}
```

Check Your Understanding (5.5)

1. What is the goal of functional decomposition?
2. If a method attempts to open a file for reading, why must its header often include a `throws IOException` clause?
3. If Method A calls Method B, and Method B throws an exception, what are the two required ways Method A can address this exception?

Practice Problems (5.5)

1. Imagine a large banking application. Outline three distinct sub-problems that could be solved using separate methods (e.g., input validation).
2. Write the method header for a `main` method that calls a static method named `saveLog` that is known to throw an `IOException`.