Chapter 6
Methods

6.1 Introduction

- Problem: Find the sum of integers from 1 to 10, from 20 to 37, and from 35 to 49, respectively. You may write the code as follows:

```java
int sum = 0;
for (int i = 1; i <= 10; i++)
    sum += i;
System.out.println("Sum from 1 to 10 is " + sum);

sum = 0;
for (int i = 20; i <= 37; i++)
    sum += i;
System.out.println("Sum from 20 to 37 is " + sum);

sum = 0;
for (int i = 35; i <= 49; i++)
    sum += i;
System.out.println("Sum from 35 to 49 is " + sum);
```

- Solutions: We can do so by defining a method and invoking it.

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

public static void main(String[] args) {
    System.out.println("Sum from 1 to 10 is " + sum(1, 10));
    System.out.println("Sum from 20 to 37 is " + sum(20, 37));
    System.out.println("Sum from 35 to 49 is " + sum(35, 49));
}
```

- A method is a collection of statements that are grouped together to perform an operation.
- Methods can be used to define reusable code and organize and simplify coding.
- You will learn how to:
  - create your own methods with or without return values,
  - invoke a method with or without parameters,
  - overload methods using the same names, and
  - apply method abstraction in the program design.
6.2 Defining a Method

- In general, a method has the following syntax:

  ```java
  Modifier returnType methodName(list of parameters) {
  // method body;
  }
  ```

- A **method** declaration consists of a method header and a method body.
- The following method finds which of two integers is bigger. The method named `max`, has two `int` parameters, `num1` and `num2`, the larger of which is returned by the method.

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

**FIGURE 6.1** A method definition consists of a method header and a method body.

- The method header specifies the **modifiers**, **return value type**, **method name**, and **parameters** of the method.
- The modifier, which is **optional**, tells the compiler how to call the method.
- The **static** modifier is used for all the methods in this chapter.
- A method may return a value. The `returnType` is the data type of the **value** the method returns.
- If the method doesn’t return a value, the `returnType` is the keyword `void`. For example, `returnType` in the main method is `void` as well as `System.out.println`.
- The **parameter list** refers to the type, order, and number of the parameters of a method. The method name and the parameter list together constitute the **method signature**. Parameters are optional; a method may contain no parameters.
- The variables defined in the method header are known as **formal parameters**.
- When a method is invoked, you pass a value to the parameter. This value is referred to as **actual parameter** or **argument**.
- The method body contains a collection of statements that define what the method does.
- A return statement using the keyword `return` is **required** for a **non-void** method to return a result.
- The method **terminates** when a return statement is executed.
6.3 Calling a Method

- To use a method, you have to call or invoke it.
- There are two ways to call a method; the choice is based on whether the method returns a value or not.
- If the method returns a value, a call to the method is usually treated as a value.

\[
\text{int larger = max(3, 4);}
\]
\[
\text{System.out.println(max(3, 4));}
\]

- If the method returns void, a call to the method must be a statement.

\[
\text{System.out.println("Welcome to Java!");}
\]

- When a program calls a method, program control is transferred to the called method.
- A called method returns control to the caller when its return statement is executed or when its method-ending closing brace is reached.

**LISTING 6.1 TestMax.java**

```java
// TestMax.java: demonstrate using the max method
public class TestMax {
    /** Main method */
    public static void main(String[] args) {
        int i = 5;
        int j = 2;
        int k = max(i, j);
        System.out.println("The maximum between " + i + " and " + j + " is " + k);
    }

    /** Return the max between two numbers */
    public static int max(int num1, int num2) {
        int result;
        if (num1 > num2)
            result = num1;
        else
            result = num2;

        return result;
    }
}
```

- The main method is just like any other method except that it is invoked by the Java interpreter.
- The main method’s header is always the same, like the one in this example, with the modifiers public and static, return value type void, method name main, and a parameter of the string[] type. String [] indicates that the parameter is an array of String.
• The statements in `main` may invoke other methods that are defined in the class that contains the `main` method or in other classes.
• The `main` method invokes `max [i, j]`, which is defined in the same class with the `main` method.
• When the `max` method is invoked, variable i’s value 5 is passed to `num1`, and variable j’s value 2 is passed to `num2` in the `max` method.
• The flow of control transfers to the `max` method. The `max` method is executed.
• When the `return` statement in the `max` method is executed, the `max` method returns the control to its caller.

FIGURE 6.2 When the `max` method is invoked, the flow of control transfers to it. Once the `max` method is finished, it returns control back to the caller.

• The variables defined in the `main` method are i, j, and k.
• The variables defined in the `max` method are `num1`, `num2` and `result`.
• The variables `num1` and `num2` are defined in the method signature and are parameters of the method. There methods are passed through method invocation.

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

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

- A return statement is required for a non-void method. The following method is logically correct, but it has a **compilation error**, because the Java compiler thinks it possible that this method does not return any value.

```java
public static int sign(int n) {
    if (n > 0) return 1;
    else if (n == 0) return 0;
    else if (n < 0) return -1;
}
```

- To fix this problem, delete `if (n < 0)` in the code.

```java
public static int sign(int n) {
    if (n > 0) return 1;
    else if (n == 0) return 0;
    else return -1;
}
```

NOTE

- One of the benefits of methods is for **sharing and reuse**. The `max` method can be invoked from any class besides `TestMax`. If you create a new class `Test`, you can invoke the `max` method using `ClassName.methodName` (i.e., `TestMax.max`).
Call Stacks

- Each time a method is invoked, the system stores parameters and local variables in an area of memory, known as a **stack**, which stores elements in **last-in first-out** fashion.
- When a method calls another method, the caller’s stack space is kept **intact**, and new space is created to handle the new method call.
- When a method finishes its work and returns to its caller, its associated space is **released**.
- The variables defined in the **main** method are **i**, **j**, and **k**.
- The variables defined in the **max** method are **num1**, **num2**, and **result**.
- The variables **num1**, **num2** are defined in the method signature and are parameters of the method.
- Their values are passed through method invocation.

**FIGURE 6.3** When the max method is invoked, the flow of control transfers to the max method. Once the max method is finished, it returns control back to the caller.
6.4 void Method Example

- This section shows how to declare and invoke a `void` method.

LISTING 6.2 TestVoidMethod.java

```java
public class TestVoidMethod {
    public static void main(String[] args) {
        System.out.print("The grade is ");
        printGrade(78.5);
        System.out.print("The grade is ");
        printGrade(59.5);
    }

    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 if (score >= 60.0) {
            System.out.println('D');
        } else {
            System.out.println('F');
        }
    }
}

The grade is C
The grade is F
```
To see the difference between a void and a value-returning method, let us redesign the printGrade method to return a value. The new method, which we call getGrade, returns the grade.

LISTING 6.3 TestReturnGradeMethod.java

```
public class TestReturnGradeMethod {
    public static void main(String[] args) {
        System.out.print("The grade is " + getGrade(78.5));
        System.out.print("\nThe grade is " + getGrade(59.5));
    }

    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 if (score >= 60.0)
            return 'D';
        else
            return 'F';
    }
}
```

The grade is C
The grade is F
6.5 Passing Arguments by Values

- When calling a method, you need to provide arguments, which must be given in the same order as their respective parameters in the method specification. This is known as parameter order association.
- You can use `nPrintln("Hello", 3)` to print “Hello” 3 times.

```java
public static void nPrintln(String message, int n) {
    for (int i = 0; i < n; i++)
        System.out.println(message);
}
```

**Caution**
- The arguments must match the parameters in order, number, and compatible type, as defined in the method signature.

**LISTING 6.5 TestPassByValue.java**

```java
// TestPassByValue.java: Demonstrate passing values to methods
public class TestPassByValue {
    /** Main method */
    public static void main(String[] args) {
        // Declare and initialize variables
        int num1 = 1;
        int num2 = 2;
        System.out.println("Before invoking the swap method, num1 is " + num1 + " and num2 is " + num2);
        // Invoke the swap method to attempt to swap two variables
        swap(num1, num2);
        System.out.println("After invoking the swap method, num1 is " + num1 + " and num2 is " + num2);
    }

    /** Swap two variables */
    public static void swap(int n1, int n2) {
        System.out.println("Inside the swap method");
        System.out.println("Before swapping n1 is " + n1 + " n2 is " + n2);
        // Swapping n1 with n2
        int temp = n1;
        n1 = n2;
        n2 = temp;
        System.out.println("After swapping n1 is " + n1 + " n2 is " + n2);
    }
}
```

- When you invoke a method with a parameter, the value of the argument is passed to the parameter. This is referred to as pass by value.
- If the argument is a variable, the value of the variable is passed to the parameter.
- The variable is not affected, regardless of the changes made to the parameter inside the method.
- Before the `swap` method is invoked, `num1` is 1 and `num2` is 2. After the `swap` method is invoked, `num1` continues to be 1 and `num2` continues to be 2.
- Their values are not swapped when the `swap` method is invoked.
- The values of the arguments `num1` and `num2` are passed to `n1` and `n2`, but `n1` and `n2` have their own memory locations independent of `num1` and `num2`.
- Therefore, changes to `n1` and `n2` do not affect the contents of `num1` and `num2`.

![Diagram showing variable and method interactions](image)

**FIGURE 6.4** The values of the variables are passed to the parameters of the method.

- The arguments and parameters may have the same name, however, **no change** occurs because the parameter is a local variable in the method with its own memory space. The local variable is **allocated** when the method is invoked, and it **disappears** when the method is returned to its caller.
6.6 Modularizing Code

- Methods can be used to reduce redundant coding and enable code reuse. Methods can also be used to modularize code and improve the quality of the program.

LISTING 6.6 GreatestCommonDivisorMethod.java

- It prompts the user to enter two integers and displays their greatest common divisor.

```java
import java.util.Scanner;

public class GreatestCommonDivisorMethod {
    /** Main method */
    public static void main(String[] args) {
        // Create a Scanner
        Scanner input = new Scanner(System.in);

        // Prompt the user to enter two integers
        System.out.print("Enter first integer: ");
        int n1 = input.nextInt();
        System.out.print("Enter second integer: ");
        int n2 = input.nextInt();

        System.out.println("The greatest common divisor for "+ n1 + " and "+ n2 + " is "+ gcd(n1, n2));
    }

    /** Return the gcd of two integers */
    public static int gcd(int n1, int n2) {
        int gcd = 1; // Initial gcd is 1
        int k = 1; // Possible gcd

        while (k <= n1 && k <= n2) {
            if (n1 % k == 0 && n2 % k == 0) {
                gcd = k; // Update gcd
                k++;
            }
        }

        return gcd; // Return gcd
    }
}
```

Enter first integer: 45
Enter second integer: 75
The greatest common divisor for 45 and 75 is 15
LISTING 6.7 PrimeNumberMethod.java

- It displays the first 50 prime numbers.

```java
public class PrimeNumberMethod {
    public static void main(String[] args) {
        System.out.println("The first 50 prime numbers are \n");
        printPrimeNumbers(50);
    }

    public static void printPrimeNumbers(int numberOfPrimes) {
        final int NUMBER_OF_PRIMES_PER_LINE = 10; // Display 10 per line
        int count = 0; // Count the number of prime numbers
        int number = 2; // A number to be tested for primeness

        // Repeatedly find prime numbers
        while (count < numberOfPrimes) {
            // Print the prime number and increase the count
            if (isPrime(number)) {
                count++; // Increase the count
                if (count % NUMBER_OF_PRIMES_PER_LINE == 0) {
                    // Print the number and advance to the new line
                    System.out.printf("%5s\n", number);
                } else
                    System.out.printf("%5s", number);
            }
            // Check if the next number is prime
            number++;
        }

        /** Check whether number is prime */
        public static boolean isPrime(int number) {
            for (int divisor = 2; divisor <= number / 2; divisor++) {
                if (number % divisor == 0) { // If true, number is not prime
                    return false; // number is not a prime
                }
            }
            return true; // number is prime
        }
    }
}
```

The first 50 prime numbers are

2  3  5  7 11 13 17 19 23 29
31 37 41 43 47 53 59 61 67 71
73 79 83 89 97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229
6.7 Case Study: Converting hexadecimals to Decimals

7F in Hexadecimal  
7 X 16^1 + 15 X 16^0 = 127 in Decimal

FFFF in Hexadecimal  
15 X 16^3 + 15 X 16^2 + 15 X 16^1 + 15 X 16^0 = 65535 in Decimal

431 in Hexadecimal  
4 X 16^2 + 3 X 16^1 + 1 X 16^0 = 1073 in Decimal

LISTING 6.7 Hex2Dec.java

```java
import java.util.Scanner;

public class Hex2Dec {
    /** Main method */
    public static void main(String[] args) {
        // Create a Scanner
        Scanner input = new Scanner(System.in);

        // Prompt the user to enter a string
        System.out.print("Enter a hex number: ");
        String hex = input.nextLine();
        System.out.println("The decimal value for hex number 
          + hex + " is " + hexToDecimal(hex.toUpperCase()));
    }

    public static int hexToDecimal(String hex) {
        int decimalValue = 0;
        for (int i = 0; i < hex.length(); i++) {
            char hexChar = hex.charAt(i);
            decimalValue = decimalValue * 16 + hexCharToDecimal(hexChar);
        }
        return decimalValue;
    }

    public static int hexCharToDecimal(char ch) {
        if (ch >= 'A' && ch <= 'F')
            return 10 + ch - 'A';
        else // ch is '0', '1', ..., or '9'
            return ch - '0';
    }
}
```

Enter a hex number: 7F
The decimal value for hex number 7F is 127

Enter a hex number: af71
The decimal value for hex number af71 is 44913
6.8 Overloading Methods

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

- If you need to find which of two floating-point numbers has the maximum value, the code above shows you just that. If you call `max` with `int` parameters, the `max` method that expects `int` parameters will be invoked. If you call `max` with `double` parameters, the `max` method that expects `double` parameters will be invoked.
- This is referred to as **method overloading**; that is, two methods have the same name but different parameters lists.
- The Java compiler determines which method is used based on the method signature.

LISTING 6.9 TestMethodOverloading.java

```java
// TestMethodOverloading.java: Demonstrate method overloading
public class TestMethodOverloading {
    /** Main method */
    public static void main(String[] args) {
        // Invoke the max method with int parameters
        System.out.println("The maximum between 3 and 4 is 
            + max(3, 4));

        // Invoke the max method with the double parameters
        System.out.println("The maximum between 3.0 and 5.4 is 
            + max(3.0, 5.4));

        // Invoke the max method with three double parameters
        System.out.println("The maximum between 3.0, 5.4, and 10.14 is 
            + max(3.0, 5.4, 10.14));
    }

    /** Return the max between two int values */
    public static int max(int num1, int num2) {
        if (num1 > num2)
            return num1;
        else
            return num2;
    }

    /** Find the max between two double values */
    public static double max(double num1, double num2) {
        if (num1 > num2)
            return num1;
        else
            return num2;
    }

    /** Return the max among three double values */
    public static double max(double num1, double num2, double num3) {
        return max(max(num1, num2), num3);
    }
}
```
The program invokes three different *max* methods that will have the same name: *max*(3, 4), *max*(3.0, 5.4), and *max*(3.0, 5.4, 10.14).

- When calling *max*(3, 4), The *max* method for finding maximum integers is invoked.
- When calling *max*(3.0, 5.4), The *max* method for finding maximum doubles is invoked.
- When calling *max*(3.0, 5.4, 10.14), The *max* method for finding maximum of three double values is invoked.

The Java compiler finds the most specific method for a method invocation. Since the method *max*(int, int) is more specific than *max*(double, double), *max*(int, int) is used to invoke *max*(3, 4).

**Overloading** methods can make programs clearer and more readable. Methods that perform closely related tasks should be given **the same name**.

**Overloaded** methods must have different parameter lists. You can’t overload methods based on different modifiers or return types.

**NOTE: Ambiguous Invocation**

- Sometimes there may be two or more possible matches for an invocation of a method, but the compiler cannot determine the most specific match. This is referred to as ambiguous invocation. **Ambiguous invocation is a compilation error.**

```java
public class AmbiguousOverloading {
    public static void main(String[] args) {
        System.out.println(max(1, 2));
    }

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

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

- Both *max* (int, double) and *max* (double, int) are possible candidates to match *max*(1, 2). Since neither of them is more specific than the other, the invocation is ambiguous.
6.9 The Scope of Variables

- A **local variable**: a variable defined inside a **method**.
- **Scope of a variable** is the part of the program where the variable can be referenced.
- The scope of a local variable **starts** from its **declaration** and continues to the **end of the block** that contains the variable.
- A local variable must be declared **before** it can be used.
- A **parameter** is actually a **local** variable. The scope of a method parameter covers the **entire** method.
- A variable declared in the initial action part of a **for** loop header has its scope in the **entire** loop. But a variable declared inside a **for** loop body has its scope limited in the loop body from its declaration and to the end of the block that contains the variable.

```java
public static void method1() {
  .
  .
  for (int i = 1; i < 10; i++) {
    .
    .
    int j;
    .
    .
    .
  } 
} 
```

- You can declare a local variable with the same name **multiple times** in different non-nesting blocks in a method, but you **cannot** declare a local variable twice in nested blocks.
- A variable can be declared multiple times in non-nested blocks, but can be declared **only** once in nesting blocks.

```java
public static void method2() {
  int i = 1;
  int sum = 0;
  for (int i = 1; i < 10; i++) {
    sum += i;
  }
}
```

**FIGURE 6.6** A variable can be declared multiple times in nonnested blocks but only once in nested blocks.
Caution

- Do not declare a variable inside a block and then use it outside the block.

```java
for (int i = 0; i < 10; i++) {
    ...
}
System.out.println(i);  // a common mistake
```

- The last statement would cause a syntax error because variable `i` is not defined outside of the `for` loop.
6.10 Case Study: Generating Random Characters

LISTING 6.10 RandomCharacter.java

```java
public class RandomCharacter {
    /** Generate a random character between ch1 and ch2 */
    public static char getRandomCharacter(char ch1, char ch2) {
        return (char)(ch1 + Math.random() * (ch2 - ch1 + 1));
    }

    /** Generate a random lowercase letter */
    public static char getRandomLowerCaseLetter() {
        return getRandomCharacter('a', 'z');
    }

    /** Generate a random uppercase letter */
    public static char getRandomUpperCaseLetter() {
        return getRandomCharacter('A', 'Z');
    }

    /** Generate a random digit character */
    public static char getRandomDigitCharacter() {
        return getRandomCharacter('0', '9');
    }

    /** Generate a random character */
    public static char getRandomCharacter() {
        return getRandomCharacter(' ', '￿');
    }
}
```

LISTING 6.11 TestRandomCharacter.java

```java
public class TestRandomCharacter {
    /** Main method */
    public static void main(String args[]) {
        final int NUMBER_OF_CHARS = 175;
        final int CHARS_PER_LINE = 25;

        // Print random characters between '!' and '~', 25 chars per line
        for (int i = 0; i < NUMBER_OF_CHARS; i++) {
            char ch = RandomCharacter.getRandomLowerCaseLetter();
            if ((i + 1) % CHARS_PER_LINE == 0)
                System.out.println(ch);
            else
                System.out.print(ch);
        }
    }
}
```
6.11 Method Abstraction and Stepwise Refinement

- *Method Abstraction* is achieved by separating the use of a method from its implementation.
- The client can use a method **without** knowing how it is implemented.
- The details of the implementation are **encapsulated** in the method and **hidden** from the client who invokes the method.
- This is known as *information hiding* or **encapsulation**.
- If you decide to change the implementation, the client program will **not** be affected, provided that you don’t change the method signature.
- The implementation of the method is hidden in a **black box** from the client.

FIGURE 6.7 The method body can be thought of as a black box that contains the detailed implementation for the method.