Chapter 4: Methods

4.1 Introduction
- A **method** is a collection of statements that are grouped together to perform an operation.
- 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,
  - write a recursive method that invokes itself, and
  - apply method abstraction in the program design.

4.2 Creating 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;
}
```

- 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 **return value type** is the data type of the value the method returns.
- If the method doesn’t return a value, the **return value type** is the keyword **void**. For example, `return value type` 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.

**Declaring a method**

```java
public static int max(int num1, int num2) {
    if (num1 > num2)
        return num1;
    else
        return num2;
}
```
4.3 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**.

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

- 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.

**EXAMPLE 4.1 TESTING THE max METHOD (Page 128)**

```java
// TestMax.java: demonstrate using the max method
global 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;
}
// Output:
// The maximum between 5 and 2 is 5
```

- 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.

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.

**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.

```
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 **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`).

### 4.3.1 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.

<table>
<thead>
<tr>
<th>Space required for the main method</th>
<th>Space required for the max method</th>
</tr>
</thead>
<tbody>
<tr>
<td>k: 5</td>
<td>result: 5</td>
</tr>
<tr>
<td>j: 2</td>
<td>num2: 2</td>
</tr>
<tr>
<td>i: 5</td>
<td>num1: 5</td>
</tr>
</tbody>
</table>

The main method is invoked.

<table>
<thead>
<tr>
<th>Space required for the main method</th>
<th>Space required for the max method</th>
</tr>
</thead>
<tbody>
<tr>
<td>k: 5</td>
<td>k: 5</td>
</tr>
<tr>
<td>j: 2</td>
<td>j: 2</td>
</tr>
<tr>
<td>i: 5</td>
<td>i: 5</td>
</tr>
</tbody>
</table>

The max method is invoked.

<table>
<thead>
<tr>
<th>Space required for the main method</th>
</tr>
</thead>
<tbody>
<tr>
<td>k: 5</td>
</tr>
<tr>
<td>j: 2</td>
</tr>
<tr>
<td>i: 5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Space required for the main method</th>
</tr>
</thead>
<tbody>
<tr>
<td>k: 5</td>
</tr>
<tr>
<td>j: 2</td>
</tr>
<tr>
<td>i: 5</td>
</tr>
</tbody>
</table>

The main method is finished.
4.4 Passing Parameters 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**. Example:

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

- You can use `nPrintln` (“Hello”, 3) to print “Hello” 3 times.

**Caution**

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

**EXAMPLE 4.2 Pass by Value (Page 131)**

- 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.

```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);
    }
}
```
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`.

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.
4.5 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.

// TestMethodOverloading.java: Demonstrate method overloading
```java
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 ");
        System.out.println(" + max(3, 4));
        // Invoke the max method with the double parameters
        System.out.println("The maximum between 3.0 and 5.4 is ");
        System.out.println(" + 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 ");
        System.out.println(" + max(3.0, 5.4, 10.14));
    }
}
```
```java
/** Return the max between two int values */
public static int max(int num1, int num2) {
    if (num1 > num2)
        return num1;
    else
        return num2;
}
```
```java
/** Find the max between two double values */
public static double max(double num1, double num2) {
    if (num1 > num2)
        return num1;
    else
        return num2;
}
```
```java
/** Return the max among three double values */
```
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.
public class ComputeTaxWithMethod {
    public static void main(String[] args) {
        // Prompt the user to enter filing status
        String statusString = JOptionPane.showInputDialog("Enter the filing status: ");
        int status = Integer.parseInt(statusString);

        // Prompt the user to enter taxable income
        String incomeString = JOptionPane.showInputDialog("Enter the taxable income: ");
        double income = Double.parseDouble(incomeString);

        // Display the result
        JOptionPane.showMessageDialog(null, "Tax is 
                (int)(computeTax(status, income) * 100) / 100.0;
            
        public static double computeTax(double income, int r1, int r2, int r3, int r4, int r5) {
            double tax = 0;
            if (income <= r1)
                tax = income * 0.10;
            else if (income <= r2)
                tax = r1 * 0.10 + (income - r1) * 0.15;
            else if (income <= r3)
                tax = r1 * 0.10 + (r2 - r1) * 0.15 + (income - r2) * 0.27;
            else if (income <= r4)
                tax = r1 * 0.10 + (r2 - r1) * 0.15 + (r3 - r2) * 0.27 + (income - r3) * 0.30;
            else if (income <= r5)
                tax = r1 * 0.10 + (r2 - r1) * 0.15 + (r3 - r2) * 0.27 + (r4 - r3) * 0.30 + (income - r4) * 0.35;
            else
                tax = r1 * 0.10 + (r2 - r1) * 0.15 + (r3 - r2) * 0.27 + (r4 - r3) * 0.30 + (r5 - r4) * 0.35 + (income - r5) * 0.386;
            return tax;
        }

        public static double computeTax(int status, double income) {
            switch (status) {
                case 0: return computeTax(income, 6000, 27950, 67700, 141250, 307050);
                case 1: return computeTax(income, 12000, 46700, 112850, 171950, 307050);
                case 2: return computeTax(income, 6000, 23350, 56425, 85975, 153525);
                case 3: return computeTax(income, 10000, 37450, 96700, 156600, 307050);
                default: return 0;
            }
        }
    }
}
4.6 The Scope of Local 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.

```
public static void method1() {
    int x = 1;
    int y = 1;
    for (int i = 1; i < 10; i++) {
        x += i;
    }
    for (int i = 1; i < 10; i++) {
        y += i;
    }
}
```

- 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.

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

Caution

- Do not declare a variable inside a block and then use it outside the block.
```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

- The last statement would cause a syntax error because variable i is not defined outside of the for loop.
4.7 Method Abstraction

- *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.

```
Optional arguments for Input

Method Signature

Method body

Optional return value

Black Box
```
4.8 The Math Class

- The Math class contains the methods needed to perform basic mathematical functions.
- Some useful methods in the Math class can be categorized as trigonometric methods, exponent methods, and service methods.
- You can use two useful double constants, PI and E (base of natural logarithm)

4.8.1 Trigonometric methods:

- sin(double a)
- cos(double a)
- tan(double a)
- acos(double a)
- asin(double a)
- atan(double a)

- Each method has a single double parameter, and its return type is double.

Examples:

<table>
<thead>
<tr>
<th>Method</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.sin(0)</td>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td>Math.sin(Math.PI / 6)</td>
<td></td>
<td>0.5</td>
</tr>
<tr>
<td>Math.sin(Math.PI / 2)</td>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>Math.cos(0)</td>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>Math.cos(Math.PI / 6)</td>
<td></td>
<td>0.866</td>
</tr>
<tr>
<td>Math.cos(Math.PI / 2)</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

4.8.2 Exponent Methods

- There are four methods related to exponents in the math class:

<table>
<thead>
<tr>
<th>Method</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp(double a)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>log(double a)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pow(double a, double b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sqrt(double a)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Examples:

<table>
<thead>
<tr>
<th>Method</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.pow(2, 3)</td>
<td></td>
<td>8.0</td>
</tr>
<tr>
<td>Math.pow(3, 2)</td>
<td></td>
<td>9.0</td>
</tr>
<tr>
<td>Math.pow(3.5, 2.5)</td>
<td></td>
<td>22.91785</td>
</tr>
<tr>
<td>Math.sqrt(4)</td>
<td></td>
<td>2.0</td>
</tr>
<tr>
<td>Math.sqrt(10.5)</td>
<td></td>
<td>3.24</td>
</tr>
</tbody>
</table>
4.8.3 The Rounding Method

- There are five methods related to rounding in the `math` class:
  
  **double ceil(double x)**  // rounded up to its nearest integer.
  // This integer is returned as a double value.
  
  **double floor(double x)**  // is rounded down to its nearest integer.
  // This integer is returned as a double value.
  
  **double rint(double x)**  // is rounded to its nearest integer.
  // If x is equally close to two integers, the even one is returned as a double.
  
  **int round(float x)**  // Return (int)Math.floor(x+0.5).
  
  **long round(double x)**  // Return (long)Math.floor(x+0.5).

```
Math.ceil(2.1)  returns 3.0
Math.ceil(2.0)  returns 2.0
Math.ceil(-2.0) returns -2.0
Math.ceil(-2.1) returns -2.0
Math.floor(2.1) returns 2.0
Math.floor(2.0) returns 2.0
Math.floor(-2.0) returns -2.0
Math.floor(-2.1) returns -3.0
Math.rint(2.1)  returns 2.0
Math.rint(2.0)  returns 2.0
Math.rint(-2.0) returns -2.0
Math.rint(-2.1) returns -2.0
Math.rint(2.5)  returns 2.0
Math.rint(-2.5) returns -2.0
Math.round(2.6f) returns 3
Math.round(2.0) returns 2
Math.round(-2.0f) returns -2
Math.round(-2.6) returns -3
```

4.8.4 The min, max, and abs Methods

- The `min` and `max` are overloaded to return the minimum and maximum numbers between two numbers.
- The `abs` is overloaded to return the absolute value of the number.

```
max(a, b) and min(a, b)  // Returns the maximum or minimum of two parameters.
abs(a)  // Returns the absolute value of the parameter.
```

Examples:

```
Math.max(2, 3)  returns 3
Math.max(2.5, 3) returns 3.0
Math.min(2.5, 3.6) returns 2.5
Math.abs(-2) returns 2
Math.abs(-2.1) returns 2.1
```
4.8.5 The random Methods

- The `random` method generates a random double value greater than or equal to 0.0 or less than 1.0 (0 <= Math.random() < 1.0).

  `random()` // Returns a random double value in the range [0.0, 1.0).

Examples:

- `(int)(Math.random() * 10)`  
  Returns a random integer between 0 and 9.

- `50 + (int)(Math.random() * 50)`  
  Returns a random integer between 50 and 99.

- `a + Math.random() * b`  
  Returns a random number between a and a + b, excluding a + b.
4.9 Case Study: Generating Random Characters

```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(' ', '￿');
    }
}

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);
        }
    }
}
```