Chapter 4 Methods

Objectives
- To create methods, invoke methods, and pass arguments to a method (§4.2-4.4).
- To use method overloading and know ambiguous overloading (§4.5).
- To determine the scope of local variables (§4.6).
- To learn the concept of method abstraction (§4.7).
- To know how to use the methods in the Math class (§4.8).
- To design and implement methods using stepwise refinement (§4.10).
- To write recursive methods (§4.11 Optional).
- To group classes into packages (§4.12 Optional).

Introducing Methods

A method is a collection of statements that are grouped together to perform an operation.

• Method signature is the combination of the method name and the parameter list.
• 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.

• A method may return a value. The returnType is the data type of the value the method returns. If the method does not return a value, the returnType is the keyword void.
For example, the returnType in the main method is void.

Calling Methods

Example 4.1 Testing the max method
This program demonstrates calling a method max to return the largest of the int values

TestMax
Calling Methods, cont.

```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 nonvoid 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.

Reuse Methods from Other Classes

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).

Call Stacks

The main method is invoked.

Space required for the main method

k:
j: 2
i: 5

The max method is invoked.

Space required for the main method
Space required for the max method

result: 5
num2: 2
num1: 5

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

The main method is finished.

Stack is empty

Space required for the main method

k: 5
j: 2
i: 5

Pass by Value

Example 4.2 Testing Pass by value

This program demonstrates passing values to the methods.

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

Suppose you invoke the method using

nPrintln("Welcome to Java", 5);
What is the output?

Suppose you invoke the method using

nPrintln("Computer Science", 15);
What is the output?
Pass by Value, cont.

The main method is invoked

The values of num1 and num2 are passed to n1 and n2. Executing swap does not affect num1 and num2.

Space required for the main method

num2: 2
num1: 1

The swap method is invoked

Space required for the swap method

temp: 1
n2: 2
n1: 1

The swap method is finished

Space required for the main method

num2: 2
num1: 1

The main method is finished

Stack is empty

Overloading Methods

Example 4.3 Overloading the max Method

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

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.

Example 4.4 Computing Taxes with Methods

Example 3.1, “Computing Taxes,” uses if statements to check the filing status and computes the tax based on the filing status. Simplify Example 3.1 using methods. Each filing status has six brackets.

The code for computing taxes is nearly same for each filing status except that each filing status has different bracket ranges. For example, the single filer status has six brackets [0, 6000], (6000, 27950], (27950, 67700], (67700, 141250], (141250, 307050], (307050, ∞), and the married file jointly status has six brackets [0, 12000], (12000, 46700], (46700, 112850], (112850, 171950], (171950, 307050], (307050, ∞).

Example 4.4 cont.

The first bracket of each filing status is taxed at 10%, the second 15%, the third 27%, the fourth 30%, the fifth 35%, and the sixth 38.6%. So you can write a method with the brackets as arguments to compute the tax for the filing status. The signature of the method is:

Example 4.4 cont.

public static double computeTax(double income, int r1, int r2, int r3, int r4, int r5)

For example, you can invoke computeTax(400000, 6000, 27950, 67700, 141250, 307050) to compute the tax for single filers with $400,000 of taxable income:
Scope of Local Variables

A local variable: a variable defined inside a method.
Scope: 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.

Scope of Local Variables, cont.

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.

// Fine with no errors
public static void correctMethod() {
    int x = 1;
    int y = 1;
    // i is declared
    for (int i = 1; i < 10; i++) {
        x += i;
    }
    // i is declared again
    for (int i = 1; i < 10; i++) {
        y += i;
    }
}

// With no errors
public static void incorrectMethod() {
    int x = 1;
    int y = 1;
    for (int i = 1; i < 10; i++) {
        int x = 0;
        x += i;
    }
}
Method Abstraction
You can think of the method body as a black box that contains the detailed implementation for the method.

Benefits of Methods
- Write a method once and reuse it anywhere.
- Information hiding. Hide the implementation from the user.
- Reduce complexity.

The Math Class
- Class constants:
  - PI
  - E
- Class methods:
  - Trigonometric Methods
  - Exponent Methods
  - Rounding Methods
  - min, max, abs, and random Methods

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

Examples:
- Math.sin(0) returns 0.0
- Math.sin(Math.PI / 6) returns 0.5
- Math.sin(Math.PI / 2) returns 1.0
- Math.cos(0) returns 1.0
- Math.cos(Math.PI / 6) returns 0.866
- Math.cos(Math.PI / 2) returns 0

Radians toRadians(90)

Exponent Methods
- exp(double a)
  Returns e raised to the power of a.
- log(double a)
  Returns the natural logarithm of a.
- pow(double a, double b)
  Returns a raised to the power of b.
- sqrt(double a)
  Returns the square root of a.

Examples:
- Math.pow(2, 3) returns 8.0
- Math.pow(3, 2) returns 9.0
- Math.pow(3.5, 2.5) returns 22.91765
- Math.sqrt(4) returns 2.0
- Math.sqrt(10.5) returns 3.24

Rounding Methods
- double ceil(double x)
  x rounded up to its nearest integer. This integer is returned as a double value.
- double floor(double x)
  x is rounded down to its nearest integer. This integer is returned as a double value.
- double rint(double x)
  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).
### Rounding Methods Examples

- `Math.ceil(2.1)` returns 3.0
- `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

### min, max, and abs

- `max(a, b)` and `min(a, b)`
  - Returns the maximum or minimum of two parameters.
- `abs(a)`
  - Returns the absolute value of the parameter.
- `random()`
  - Returns a random double value in the range [0.0, 1.0).

#### 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

### The random Methods

Generates a random double value greater than or equal to 0.0 and less than 1.0 (0 <= Math.random() < 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.

**In general,**

\[
a + Math.random() * b
\]

returns a random number between \(a\) and \(a + b\), excluding \(a + b\).

**Case Study: Generating Random Characters**

Computer programs process numerical data and characters. You have seen many examples involving numerical data. It is also important to understand characters and how to process them.

As introduced in Section 2.9, each character has a unique Unicode between 0 and FFFF in hexadecimal (65535 in decimal). To generate a random character, you can use the following expression:

\[
(int)((int)'a' + Math.random() * ((int)'z' - (int)'a' + 1))
\]

In general,

\[
(int)((int)'a' + Math.random() * ((int)'z' - (int)'a' + 1))
\]

returns a random integer between \(\text{(int)'a'}\) and \(\text{(int)'z'}\), excluding \(\text{(int)'z'}\).

**Case Study: Generating Random Characters, cont.**

Now let us consider how to generate a random lowercase letter. The Unicode for lowercase letters is a consecutive sequence of integers starting from the Unicode for 'a', then for 'b', 'c', ..., and 'z'. The Unicode for 'a' is

\[
(int)'a'
\]

So, a random integer between \(\text{(int)'a'}\) and \(\text{(int)'z'}\) is

\[
(int)((int)'a' + Math.random() * ((int)'z' - (int)'a' + 1))
\]

**Case Study: Generating Random Characters, cont.**

Now let us consider how to generate a random lowercase letter. The Unicode for lowercase letters is a consecutive sequence of integers starting from the Unicode for 'a', then for 'b', 'c', ..., and 'z'. The Unicode for 'a' is

\[
(int)'a'
\]

So, a random integer between \(\text{(int)'a'}\) and \(\text{(int)'z'}\) is

\[
(int)((int)'a' + Math.random() * ((int)'z' - (int)'a' + 1))
\]
Case Study: Generating Random Characters, cont.

As discussed in Section 2.9.4, all numeric operators can be applied to the char operands. The char operand is cast into a number if the other operand is a number or a character. So, the preceding expression can be simplified as follows:

\[ 'a' + \text{Math.random()} \times ('z' - 'a' + 1) \]

So a random lowercase letter is

\[(\text{char})'a' + \text{Math.random()} \times ('z' - 'a' + 1)]

Case Study: Generating Random Characters, cont.

To generalize the foregoing discussion, a random character between any two characters ch1 and ch2 with ch1 < ch2 can be generated as follows:

\[(\text{char})(\text{ch1} + \text{Math.random()} \times (\text{ch2} - \text{ch1} + 1))\]

The RandomCharacter Class

```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(' ', '￿');
    }
}
```

Stepwise Refinement (Optional)

The concept of method abstraction can be applied to the process of developing programs. When writing a large program, you can use the “divide and conquer” strategy, also known as stepwise refinement, to decompose it into subproblems. The subproblems can be further decomposed into smaller, more manageable problems.

PrintCalendar Example

Let us use the PrintCalendar example demonstrate the stepwise refinement approach.

Design Diagram

```
input

readInput

printMonth

printMonthTitle

printMonthBody

getTotalNumberOfDays

getDayName

isLeapYear

printCalendar
```

```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(' ', '￿');
    }
```
Implementation: Top-Down

Top-down approach is to implement one method in the structure chart at a time from the top to the bottom. Stubs can be used for the methods waiting to be implemented. A stub is a simple but incomplete version of a method. The use of stubs enables you to test invoking the method from a caller. Implement the main method first and then use a stub for the printMonth method. For example, let printMonth display the year and the month in the stub. Thus, your program may begin like this:

A Skeleton for printCalendar

Implementation: Bottom-Up

Bottom-up approach is to implement one method in the structure chart at a time from the bottom to the top. For each method implemented, write a test program to test it. Both top-down and bottom-up methods are fine. Both approaches implement the methods incrementally and help to isolate programming errors and makes debugging easy. Sometimes, they can be used together.

Optional

Recursion

Example 4.5 Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

Factorial(3) = 3 * factorial(2) = 3 * (2 * factorial(1)) = 3 * (2 * (1 * factorial(0))) = 3 * (2 * 1) = 3 * 2 = 6

ComputeFactorial

Example 4.5 Computing Factorial, cont.

Example 4.6 Computing Fibonacci Numbers

Fibonacci series: 0 1 1 2 3 5 8 13 21 34 55 89...

fib(0) = 0;
fib(1) = 1;

fib(index) = fib(index -1) + fib(index -2); index >=2

fib(3) = fib(2) + fib(1) = (fib(1) + fib(0)) + fib(1) = (1 + 0) + fib(1) = 1 + fib(1) = 1 + 1 = 2

ComputeFibonacci
Fibonacci Numbers, cont.

fib(4) = fib(3) + fib(2)
fib(3) = fib(2) + fib(1)
fib(2) = fib(1) + fib(0)
fib(1) = 1
fib(2) = fib(1) + fib(0)
1: call fib(3)
2: call fib(2)
3: call fib(1)
4: return fib(1)
7: return fib(2)
5: call fib(0)
6: return fib(0)
8: call fib(1)
9: return fib(1)
10: return fib(3)
11: call fib(2)
16: return fib(2)
12: call fib(1)
13: return fib(1)
14: return fib(0)
15: return fib(0)

Towers of Hanoi

Example 4.7 Solving the Towers of Hanoi Problem

Solve the towers of Hanoi problem.

Exercise 4.15 GCD

gcd(2, 3) = 1
gcd(2, 10) = 2
gcd(25, 35) = 5
gcd(205, 301) = 5
gcd(m, n)

Approach 1: Brute-force, start from min(n, m) down to 1,
to check if a number is common divisor for both m and
n, if so, it is the greatest common divisor.

Approach 2: Euclid's algorithm

Approach 3: Recursive method

Approach 2: Euclid's algorithm

```
// Get absolute value of m and n;
t1 = Math.abs(m); t2 = Math.abs(n);
// r is the remainder of t1 divided by t2;
while (r != 0) {
    t1 = t2;
    t2 = r;
    r = t1 % t2;
}
// When r is 0, t2 is the greatest common
divisor between t1 and t2
return t2;
```

Approach 3: Recursive Method

```
gcd(m, n) = n if m % n = 0;
gcd(m, n) = gcd(n, m % n); otherwise;
```
### Package

There are three reasons for using packages:

1. **To avoid naming conflicts.** When you develop reusable classes to be shared by other programmers, naming conflicts often occur. To prevent this, put your classes into packages so that they can be referenced through package names.
2. **To distribute software conveniently.** Packages group related classes so that they can be easily distributed.
3. **To protect classes.** Packages provide protection so that the protected members of the classes are accessible to the classes in the same package, but not to the external classes.

### Package-Naming Conventions

Packages are hierarchical, and you can have packages within packages. For example, `java.lang.Math` indicates that `Math` is a class in the package `java.lang` and that `java.lang` is a package in the package `java`.

Levels of nesting can be used to ensure the uniqueness of package names.

Choosing a unique name is important because your package may be used on the Internet by other programs. Java designers recommend that you use your Internet domain name in reverse order as a package prefix. Since Internet domain names are unique, this prevents naming conflicts. Suppose you want to create a package named `mypackage` on a host machine with the Internet domain name `prenhall.com`. To follow the naming convention, you would name the entire package `com.prenhall.mypackage`. By convention, package names are all in lowercase.

### Package Directories

Java expects one-to-one mapping of the package name and the file system directory structure. For the package named `com.prenhall.mypackage`, you must create a directory, as shown in the figure. In other words, a package is actually a directory that contains the bytecode of the classes.

```
com.prenhall.mypackage
```

The `com` directory does not have to be the root directory. In order for Java to know where your package is in the file system, you must modify the environment variable `classpath` so that it points to the directory in which your package resides.

### Setting classpath Environment

The `com` directory does not have to be the root directory. In order for Java to know where your package is in the file system, you must modify the environment variable `classpath` so that it points to the directory in which your package resides.

Suppose the `com` directory is under `c:/book`. The following line adds `c:/book` into the `classpath`:

```
classpath=.;c:/book;
```

The period (`.`) indicating the current directory is always in `classpath`. The directory `c:/book` is in `classpath` so that you can use the package `com.prenhall.mypackage` in the program.

### Setting Paths in JBuilder

An IDE such as JBuilder uses the source directory path to specify where the source files are stored and uses the class directory path to specify where the compiled class files are stored.

A source file must be stored in a package directory under the source directory path. For example, if the source directory is `c:/mysource` and the package statement in the source code is package `com.prenhall.mypackage`, then the source code file must be stored in `c:/mysource/com/prenhall/mypackage`.

A class file must be stored in a package directory under the class directory path. For example, if the class directory is `c:/myclass` and the package statement in the source code is package `com.prenhall.mypackage`, then the class file must be stored in `c:/myclass/com/prenhall/mypackage`.

### Putting Classes into Packages

Every class in Java belongs to a package. The class is added to the package when it is compiled. All the classes that you have used so far in this book were placed in the current directory (a default package) when the Java source programs were compiled. To put a class in a specific package, you need to add the following line as the first noncomment and nonblank statement in the program:

```
package packagename;
```
Example 4.8 Putting Classes into Packages

Problem
This example creates a class named `Format` and places it in the package `com.prenhall.mypackage`. The `Format` class contains the `format(number, numOfDecimalDigits)` method that returns a new number with the specified number of digits after the decimal point. For example, `format(10.3422345, 2)` returns 10.34, and `format(-0.343434, 3)` returns -0.343.

Solution
1. Create `Format.java` as follows and save it into `c:\book\com\prenhall\mypackage`.
   ```java
   // Format.java: Format number.
   package com.prenhall.mypackage;
   public class Format {
       public static double format(
           double number, int numOfDecimalDigits) {
           return Math.round(number * Math.pow(10, numOfDecimalDigits)) / Math.pow(10, numOfDecimalDigits);
       }
   }
   ```
2. Compile `Format.java`. Make sure `Format.class` is in `c:\book\com\prenhall\mypackage`.

Example 4.9 Using Packages

Problem
This example shows a program that uses the `Format` class in the `com.prenhall.mypackage.mypackage` package.

Solution
1. Create `TestFormatClass.java` as follows and save it into `c:\book`.
   ```java
   // TestFormatClass.java: Demonstrate using the Format class
   import com.prenhall.mypackage.Format;
   public class TestFormatClass {
       /** Main method */
       public static void main(String[] args) {
           System.out.println(Format.format(10.3422345, 2));
           System.out.println(Format.format(-0.343434, 3));
       }
   }
   ```

Using Classes from Packages

There are two ways to use classes from a package.
- One way is to use the fully qualified name of the class. For example, the fully qualified name for `JOptionPane` is `javax.swing.JOptionPane`. For `Format` in the preceding example, it is `com.prenhall.mypackage.Format`. This is convenient if the class is used a few times in the program.
- The other way is to use the import statement. For example, to import all the classes in the `javax.swing` package, you can use `import javax.swing.*;`

An import that uses a `*` is called an import on demand declaration. You can also import a specific class. For example, this statement imports `javax.swing.JOptionPane`:
```java
import javax.swing.JOptionPane;
```

The information for the classes in an imported package is not read in at compile time or runtime unless the class is used in the program. The import statement simply tells the compiler where to locate the classes.