

GLOBAL
EDITION



Introduction to Java™ Programming

Brief Version

ELEVENTH EDITION

Y. Daniel Liang



Digital Resources for Students

Your new textbook provides 12-month access to digital resources that may include VideoNotes (step-by-step video tutorials on programming concepts), source code, web chapters, quizzes, and more. Refer to the preface in the textbook for a detailed list of resources.

Follow the instructions below to register for the Companion Website for Daniel Liang's *Introduction to Java™ Programming, Brief Version*, Eleventh Edition, Global Edition.

1. Go to www.pearsonglobaleditions.com/liang
2. Enter the title of your textbook or browse by author name.
3. Click Companion Website.
4. Click Register and follow the on-screen instructions to create a login name and password.

ISSLJB - FLUFF - ALIEN - PAREU - BEGUN - OOSSE

Use the login name and password you created during registration to start using the digital resources that accompany your textbook.

IMPORTANT:

This prepaid subscription does not include access to MyProgrammingLab, which is available at www.myprogramminglab.com for purchase.

This access code can only be used once. This subscription is valid for 12 months upon activation and is not transferable. If the access code has already been revealed it may no longer be valid.

For technical support go to <https://support.pearson.com/getsupport>

You may be wondering why the method `max(double, double)` is not invoked for the call `max(3, 4)`. Both `max(double, double)` and `max(int, int)` are possible matches for `max(3, 4)`. The Java compiler finds the method that best matches a method invocation. Since the method `max(int, int)` is a better match for `max(3, 4)` than `max(double, double)`, `max(int, int)` is used to invoke `max(3, 4)`.



Tip

Overloading methods can make programs clearer and more readable. Methods that perform the same function with different types of parameters should be given the same name.



Note

Overloaded methods must have different parameter lists. You cannot overload methods based on different modifiers or return types.



Note

Sometimes there are two or more possible matches for the invocation of a method, but the compiler cannot determine the most specific match. This is referred to as *ambiguous invocation*. Ambiguous invocation causes a compile error. Consider the following code:

ambiguous invocation

```
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)`. Because neither is more specific than the other, the invocation is ambiguous, resulting in a compile error.

6.8.1 What is method overloading? Is it permissible to define two methods that have the same name but different parameter types? Is it permissible to define two methods in a class that have identical method names and parameter lists, but different return value types or different modifiers?



6.8.2 What is wrong in the following program?

```
public class Test {
    public static void method(int x) {
    }

    public static int method(int y) {
```

```
        return y;
    }
}
```

6.8.3 Given two method definitions,

```
public static double m(double x, double y)

public static double m(int x, double y)
```

tell which of the two methods is invoked for:

- a. `double z = m(4, 5);`
- b. `double z = m(4, 5.4);`
- c. `double z = m(4.5, 5.4);`



6.9 The Scope of Variables

The scope of a variable is the part of the program where the variable can be referenced.

scope of variables
local variable

Section 2.5 introduced the scope of a variable. This section discusses the scope of variables in detail. A variable defined inside a method is referred to as a *local variable*. 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 and assigned a value 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. However, a variable declared inside a `for`-loop body has its scope limited in the loop body from its declaration to the end of the block that contains the variable, as shown in Figure 6.5.

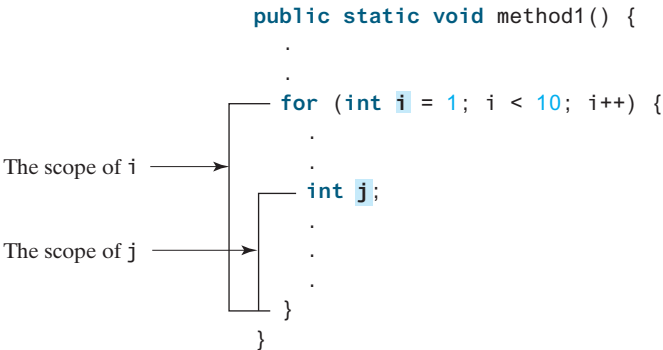


FIGURE 6.5 A variable declared in the initial-action part of a `for`-loop header has its scope in the entire loop.

You can declare a local variable with the same name in different blocks in a method, but you cannot declare a local variable twice in the same block or in nested blocks, as shown in Figure 6.6.

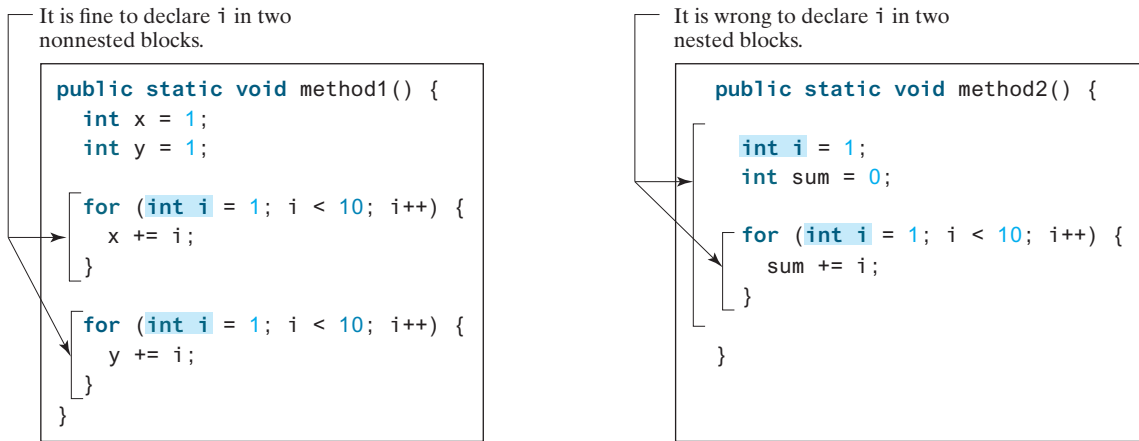


FIGURE 6.6 A variable can be declared multiple times in nonnested blocks, but only once in nested blocks.



Caution

A common mistake is to declare a variable in a **for** loop and then attempt to use it outside the loop. As shown in the following code, **i** is declared in the **for** loop, but it is accessed from the outside of the **for** loop, which causes a syntax error.

```

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

System.out.println(i); // Causes a syntax error on i

```

The last statement would cause a syntax error, because variable **i** is not defined outside of the **for** loop.

6.9.1 What is a local variable?

6.9.2 What is the scope of a local variable?



6.10 Case Study: Generating Random Characters

A character is coded using an integer. Generating a random character is to generate an integer.



Computer programs process numerical data and characters. You have seen many examples that involve numerical data. It is also important to understand characters and how to process them. This section presents an example of generating random characters.

As introduced in Section 4.3, every character has a unique Unicode between **0** and **FFFF** in hexadecimal (**65535** in decimal). To generate a random character is to generate a random integer between **0** and **65535** using the following expression (note since **0 <= Math.random() < 1.0**, you have to add **1** to **65535**):

```
(int)(Math.random() * (65535 + 1))
```

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

```
(int)'a'
```

Thus, a random integer between **(int)'a'** and **(int)'z'** is

```
(int)((int)'a' + Math.random() * ((int)'z' - (int)'a' + 1))
```

As discussed in Section 4.3.3, 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. Therefore, the preceding expression can be simplified as follows:

```
'a' + Math.random() * ('z' - 'a' + 1)
```

and a random lowercase letter is

```
(char)('a' + Math.random() * ('z' - 'a' + 1))
```

Hence, a random character between any two characters `ch1` and `ch2` with `ch1 < ch2` can be generated as follows:

```
(char)(ch1 + Math.random() * (ch2 - ch1 + 1))
```

This is a simple but useful discovery. Listing 6.10 defines a class named `RandomCharacter` with overloaded methods to get a certain type of random character. You can use these methods in your future projects.

LISTING 6.10 RandomCharacter.java

```

1 public class RandomCharacter {
2     /** Generate a random character between ch1 and ch2 */
3     public static char getRandomCharacter(char ch1, char ch2) {
4         return (char)(ch1 + Math.random() * (ch2 - ch1 + 1));
5     }
6
7     /** Generate a random lowercase letter */
8     public static char getRandomLowerCaseLetter() {
9         return getRandomCharacter('a', 'z');
10    }
11
12    /** Generate a random uppercase letter */
13    public static char getRandomUpperCaseLetter() {
14        return getRandomCharacter('A', 'Z');
15    }
16
17    /** Generate a random digit character */
18    public static char getRandomDigitCharacter() {
19        return getRandomCharacter('0', '9');
20    }
21
22    /** Generate a random character */
23    public static char getRandomCharacter() {
24        return getRandomCharacter('\u0000', '\uFFFF');
25    }
26 }

```

getRandomCharacter

getRandomLowerCaseLetter()

getRandomUpperCaseLetter()

getRandomDigitCharacter()

getRandomCharacter()

Listing 6.11 gives a test program that displays 175 random lowercase letters.

LISTING 6.11 TestRandomCharacter.java

```

1 public class TestRandomCharacter {
2     /** Main method */
3     public static void main(String[] args) {
4         final int NUMBER_OF_CHARS = 175;
5         final int CHARS_PER_LINE = 25;
6
7         // Print random characters between 'a' and 'z', 25 chars per line
8         for (int i = 0; i < NUMBER_OF_CHARS; i++) {
9             char ch = RandomCharacter.getRandomLowerCaseLetter();
10            if ((i + 1) % CHARS_PER_LINE == 0)
11                System.out.println(ch);

```

constants

lowercase letter

```

12         else
13             System.out.print(ch);
14     }
15 }
16 }

```

```

gmjssohezfkgtazqgmswfc1rao
pnrunulnwmazt1fjedmpchcif
1alqdgivxkxpbzulrmqmbhikr
lbnrjlsopfxahssqhwuuljvbe
xbhdotzhpehbqmuwsfktwsoli
cbuwkzgxpmtzihgatdslvbwbz
bfesoklwbhnooygiigzdxuqni

```



Line 9 invokes `getRandomLowerCaseLetter()` defined in the `RandomCharacter` class. Note `getRandomLowerCaseLetter()` does not have any parameters, but you still have to use the parentheses when defining and invoking the method.

parentheses required

6.1.1 Method Abstraction and Stepwise Refinement

The key to developing software is to apply the concept of abstraction.

You will learn many levels of abstraction from this book. *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 also known as *information hiding* or *encapsulation*. If you decide to change the implementation, the client program will not be affected, provided that you do not change the method signature. The implementation of the method is hidden from the client in a “black box,” as shown in Figure 6.7.



VideoNote

Stepwise refinement

method abstraction
information hiding

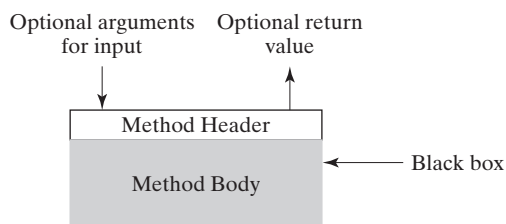


FIGURE 6.7 The method body can be thought of as a black box that contains the detailed implementation for the method.

You have already used the `System.out.print` method to display a string and the `max` method to find the maximum number. You know how to write the code to invoke these methods in your program, but as a user of these methods, you are not required to know how they are implemented.

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.

divide and conquer
stepwise refinement

Suppose that you write a program that displays the calendar for a given month of the year. The program prompts the user to enter the year and the month, and then displays the entire calendar for the month, as presented in the following sample run:



Enter full year (e.g., 2012): 2012

Enter month as number between 1 and 12: 3

March 2012

Sun	Mon	Tue	Wed	Thu	Fri	Sat
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

Let us use this example to demonstrate the divide-and-conquer approach.

6.11.1 Top-Down Design

How would you get started on such a program? Would you immediately start coding? Beginning programmers often start by trying to work out the solution to every detail. Although details are important in the final program, concern for detail in the early stages may block the problem-solving process. To make problem solving flow as smoothly as possible, this example begins by using method abstraction to isolate details from design and only later implements the details.

For this example, the problem is first broken into two subproblems: get input from the user, and print the calendar for the month. At this stage, you should be concerned with what the subproblems will achieve, not with how to get input and print the calendar for the month. You can draw a structure chart to help visualize the decomposition of the problem (see Figure 6.8a).

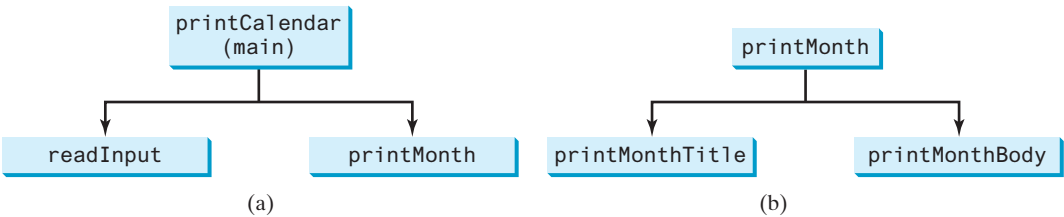


FIGURE 6.8 The structure chart shows the `printCalendar` problem is divided into two subproblems, `readInput` and `printMonth` in (a), and `printMonth` is divided into two smaller subproblems, `printMonthTitle` and `printMonthBody` in (b).

You can use `Scanner` to read input for the year and the month. The problem of printing the calendar for a given month can be broken into two subproblems: print the month title, and print the month body, as shown in Figure 6.8b. The month title consists of three lines: month and year, a dashed line, and the names of the seven days of the week. You need to get the month name (e.g., January) from the numeric month (e.g., 1). This is accomplished in `getMonthName` (see Figure 6.9a).

In order to print the month body, you need to know which day of the week is the first day of the month (`getStartDay`) and how many days the month has (`getNumberOfDaysInMonth`),