6.9 Case Study: Secure Random-Number Generation

We now take a brief diversion into a popular type of programming application—simulation and game playing. In this and the next section, we develop a game-playing program with multiple methods. The program uses most of the control statements presented thus far in the book and introduces several new programming concepts.

The **element of chance** can be introduced in a program via an object of class `Secure-Random` (package `java.security`). Such objects can produce random `boolean`, `byte`, `float`, `double`, `int`, `long` and Gaussian values. In the next several examples, we use objects of class `SecureRandom` to produce random values.

Moving to Secure Random Numbers

Recent editions of this book used Java's `Random` class to obtain "random" values. This class produced *deterministic* values that could be *predicted* by malicious programmers.

`Secure-Random` objects produce **nondeterministic random numbers** that *cannot* be predicted.

Deterministic random numbers have been the source of many software security breaches. Most programming languages now have library features similar to `SecureRandom` for producing nondeterministic random numbers to help prevent such problems. From this point forward in the text, when we refer to “random numbers” we mean “secure random numbers.”

9

Software Engineering Observation 6.7

For developers concerned with building increasingly secure applications, Java 9 enhances `SecureRandom`*’s capabilities as defined by JEP 273.*

Creating a `SecureRandom` Object

A new secure random-number generator object can be created as follows:

```
SecureRandom randomNumbers = new SecureRandom();
```

It can then be used to generate random values—we discuss only random `int` values here. For more information on the `SecureRandom` class, see

```
http://docs.oracle.com/javase/8/docs/api/java/securit
```

Obtaining a Random `int` Value

Consider the following statement:

```
int randomValue = randomNumbers.nextInt();
```

`SecureRandom` method `nextInt` generates a random `int` value. If it truly produces values *at random*, then every value in the range should have an *equal chance* (or probability) of being chosen each time `nextInt` is called.

Changing the Range of Values Produced By

`nextInt`

The range of values produced by method `nextInt` generally differs from the range of values required in a particular Java application. For example, a program that simulates coin tossing might require only 0 for “heads” and 1 for “tails.” A program that simulates the rolling of a six-sided die might require random integers in the range 1–6. A program that randomly predicts the next type of spaceship (out of four possibilities) that will fly across the horizon in a video game might require random integers in the range 1–4. For cases like these, class `SecureRandom` provides another version of method `nextInt` that receives an `int` argument and returns a value from 0 up to, but not including, the argument’s value. For example, for coin tossing, the following statement returns 0 or 1.

```
int randomValue = randomNumbers.nextInt(2);
```

Rolling a Six-Sided Die

To demonstrate random numbers, let’s develop a program that simulates 20 rolls of a six-sided die and displays the value of each roll. We begin by using `nextInt` to produce random values in the range 0–5, as follows:

```
int face = randomNumbers.nextInt(6);
```

The argument `6`—called the **scaling factor**—represents the number of unique values that `nextInt` should produce (in this case six—0, 1, 2, 3, 4 and 5). This manipulation is called **scaling** the range of values produced by `SecureRandom` method `nextInt`.

A six-sided die has the numbers 1–6 on its faces, not 0–5. So we **shift** the range of numbers produced by adding a **shifting value**—in this case 1—to our previous result, as in

```
int face = 1 + randomNumbers.nextInt(6);
```

The shifting value (`1`) specifies the *first* value in the desired range of random integers. The preceding statement assigns `face` a random integer in the range 1–6.

Rolling a Six-Sided Die 20 Times

Figure 6.6 shows two sample outputs which confirm that the results of the preceding calculation are integers in the range 1–6, and that each run of the program can produce a *different* sequence of random numbers. Line 3 imports class `SecureRandom` from the `java.security` package. Line 8 creates the `SecureRandom` object `randomNumbers` to produce random values. Line 13 executes 20 times in a loop to roll the die. The `if` statement (lines 18–20) in the loop starts a

new line of output after every five numbers to create a neat, five-column format.

```
1 // Fig. 6.6: RandomIntegers.java
2 // Shifted and scaled random integers.
3 import java.security.SecureRandom; // program uses
4
5 public class RandomIntegers {
6    public static void main(String[] args) {
7       // randomNumbers object will produce secure
8       SecureRandom randomNumbers = new SecureRando
9
10       // loop 20 times
11       for (int counter = 1; counter <= 20; counter
12          // pick random integer from 1 to 6
13          int face = 1 + randomNumbers.nextInt(6);
14
15          System.out.printf("%d  ", face); // disp
16
17          // if counter is divisible by 5, start a
18             if (counter % 5 == 0) {
19                System.out.println();
20             }
21          }
22    }
23 }
```

```
1  5  3  6  2
5  2  6  5  2
4  4  4  2  6
3  1  6  2  2
```

```
6  5  4  2  6
1  2  5  1  3
```

```
6  3  2  2  1
6  4  2  6  4
```

Fig. 6.6

Shifted and scaled random integers.

Rolling a Six-Sided Die 60,000,000 Times

To show that the numbers produced by `nextInt` occur with approximately equal likelihood, let's simulate 60,000,000 rolls of a die with the application in Fig. 6.7. Each integer from 1 to 6 should appear approximately 10,000,000 times. Note in line 18 that we used the _ digit separator to make the `int` value `60_000_000` more readable. Recall that you cannot separate digits with commas. For example, if you replace the `int` value `60_000_000` with `60,000,000`, the JDK 8 compiler generates several compilation errors throughout the `for` statement's header (line 18). Note that this example might take several seconds to execute—see our note about `SecureRandom` performance following the example.

Fig. 6.7

Roll a six-sided die 60,000,000 times.

```
1 // Fig. 6.7: RollDie.java
2 // Roll a six-sided die 60,000,000 times.
3 import java.security.SecureRandom;
4
5 public class RollDie {
6    public static void main(String[] args) {
7       // randomNumbers object will produce secure
8       SecureRandom randomNumbers = new SecureRando
9
10      int frequency1 = 0; // count of 1s rolled
11      int frequency2 = 0; // count of 2s rolled
12      int frequency3 = 0; // count of 3s rolled
13      int frequency4 = 0; // count of 4s rolled
14      int frequency5 = 0; // count of 5s rolled
15      int frequency6 = 0; // count of 6s rolled
16
17      // tally counts for 60,000,000 rolls of a di
18      for (int roll = 1; roll <= 60_000_000; roll+
19         int face = 1 + randomNumbers.nextInt(6);
20
21         // use face value 1-6 to determine which
22         switch (face) {
23            case 1:
24               ++frequency1; // increment the 1s c
25               break;
26            case 2:
27               ++frequency2; // increment the 2s c
28               break;
29            case 3:
30               ++frequency3; // increment the 3s c
31               break;
32            case 4:
33               ++frequency4; // increment the 4s c
34               break;
35            case 5:
36               ++frequency5; // increment the 5s c
```

```
            37                 break;
             38             case 6:
  39                ++frequency6; // increment the 6s c
            40                 break;
                 41           }
                  42       }
                      43
  44       System.out.println("Face\tFrequency"); // ou
  45       System.out.printf("1\t%d%n2\t%d%n3\t%d%n4\t%
  46          frequency1, frequency2, frequency3, frequ
       47           frequency5, frequency6);
                    48    }
                     49 }
```

Face	Frequency
1	10001086
2	10000185
3	9999542
4	9996541
5	9998787
6	10003859

Face	Frequency
1	10003530
2	9999925
3	9994766
4	10000707
5	9998150
6	10002922

As the sample outputs show, scaling and shifting the values produced by `nextInt` enables the program to simulate rolling a six-sided die. The application uses nested control statements (the `switch` is nested inside the `for`) to determine the number of times each side of the die appears. The `for` statement (lines 18–42) iterates 60,000,000 times. During each iteration, line 19 produces a random value from 1 to 6. That value is then used as the controlling expression (line 22) of the `switch` statement (lines 22–41). Based on the `face` value, the `switch` statement increments one of the six counter variables during each iteration of the loop. This `switch` statement has no `default` case, because we have a `case` for every possible die value that the expression in line 19 could produce. Run the program, and observe the results. As you'll see, every time you run this program, it produces *different* results.

8

When we study arrays in Chapter 7, we'll show an elegant way to replace the entire `switch` statement in this program with a *single* statement. Then, when we study Java SE 8's functional programming capabilities in Chapter 17, we'll show how to replace the loop that rolls the dice, the `switch` statement *and* the statement that displays the results with a *single* statement!

A Note about

SecureRandom Performance

Using SecureRandom instead of Random to achieve higher levels of security incurs a significant performance penalty. For "casual" applications, you might want to use class Random from package java.util—simply replace SecureRandom with Random.

Generalized Scaling and Shifting of Random Numbers

Previously, we simulated the rolling of a six-sided die with the statement

```
int face = 1 + randomNumbers.nextInt(6);
```

This statement always assigns to variable face an integer in the range 1 ≤ face ≤ 6. The *width* of this range (i.e., the number of consecutive integers in the range) is 6, and the *starting number* in the range is 1. In the preceding statement, the width of the range is determined by the number 6 that's passed as an argument to SecureRandom method nextInt, and the starting number of the range is the number 1 that's added to randomNumbers.next-Int(6). We

can generalize this result as

```
int number = shiftingValue + randomNumbers.nextInt(sc
```

where *shiftingValue* specifies the *first number* in the desired range of consecutive integers and *scalingFactor* specifies *how many numbers* are in the range.

It's also possible to choose integers at random from sets of values other than ranges of consecutive integers. For example, to obtain a random value from the sequence 2, 5, 8, 11 and 14, you could use the statement

```
int number = 2 + 3 * randomNumbers.nextInt(5);
```

In this case, `randomNumbers.nextInt(5)` produces values in the range 0–4. Each value produced is multiplied by 3 to produce a number in the sequence 0, 3, 6, 9 and 12. We add 2 to that value to *shift* the range of values and obtain a value from the sequence 2, 5, 8, 11 and 14. We can generalize this result as

```
int number = shiftingValue +
   differenceBetweenValues * randomNumbers.nextInt(sc
```

where *shiftingValue* specifies the first number in the desired range of values, *difference-BetweenValues* represents the *constant difference* between consecutive numbers in the

sequence and *scalingFactor* specifies how many numbers are in the range.