

Generating Random Numbers

In this appendix, you will:

- © Understand random numbers generated by computers
- © Use the `Math.random()` method
- © Use the `Random` class

Understanding Random Numbers Generated by Computers

1022

A **random number** is a number whose value cannot be predicted. Many types of programs use random numbers. For example, simulations that predict phenomena such as urban traffic patterns, crop production, and weather systems typically use random numbers. You might want to use random numbers to change your screen's appearance; for example, screen savers often use random numbers so that a changing pattern remains interesting.

Random numbers are also used in many computer game applications. When you play games with human opponents, their choices are often unpredictable (and sometimes even irrational). Computers usually are predictable and rational, so when you play a game against a computer opponent, you frequently need to generate random numbers. For example, a guessing game would not be very interesting if you were asked to guess the same number every time you played.

Most computer programming languages, including Java, come with built-in methods that generate random numbers. The random numbers are calculated based on a starting value, called a **seed**. The random numbers generated using these methods are not truly random; they are **pseudorandom** in that they produce the same set of numbers whenever the seed is the same. Therefore, if you seed a random-number generator with a constant, you always receive the same sequence of values. Many computer programs use the time of day as a random number-generating seed. For game applications, this method works well, as a player is unlikely to reset his computer's clock and attempt to replay a game beginning at exactly the same moment in time.



For applications in which randomness is more crucial than in game playing, you can use other methods (such as using the points in time at which a radioactive source decays) to generate truly random starting numbers.

There are two approaches to generating random numbers in Java. Both techniques are explained in this appendix and summarized in Table D-1.

Method/Class	Advantages
Math.random() method	You do not need to create an object You do not need to understand constructors and multiple methods
Random class and its methods	You can generate numbers in the format you need without arithmetic manipulation You can create reproducible results if necessary

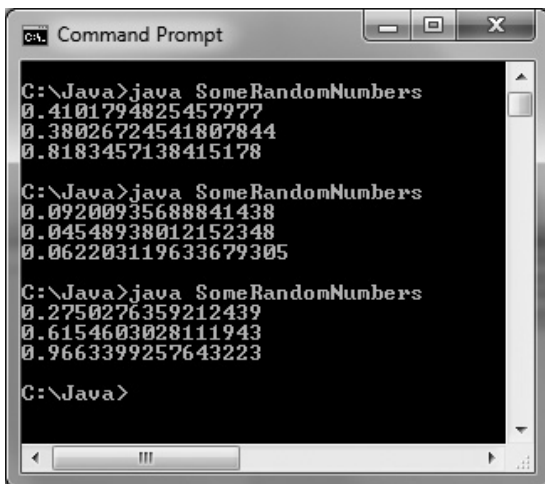
Table D-1 Generating random numbers in Java

Using the Math.random() Method

Java's Math class provides a random() method that returns a double value in the range of 0.0 up to, but not including, 1.0. For example, the application in Figure D-1 generates three random numbers and displays them. Figure D-2 shows three successive executions of the program.

```
public class SomeRandomNumbers
{
    public static void main (String[] args)
    {
        double ran;
        ran = Math.random();
        System.out.println(ran);
        ran = Math.random();
        System.out.println(ran);
        ran = Math.random();
        System.out.println(ran);
    }
}
```

Figure D-1 The SomeRandomNumbers class



```
C:\Java>java SomeRandomNumbers
0.4101794825457977
0.38026724541807844
0.8183457138415178

C:\Java>java SomeRandomNumbers
0.09200935688841438
0.04548938012152348
0.062203119633679305

C:\Java>java SomeRandomNumbers
0.2750276359212439
0.6154603028111943
0.9663399257643223

C:\Java>
```

Figure D-2 Three executions of the SomeRandomNumbers program

The values displayed in Figure D-2 appear to be random, but they are not typical of the values you need in a game-playing program. Usually, you need a relatively small number of whole values. For example, a game that involves a coin flip might need only two values to represent heads or tails, and a dice game might need only six values to represent rolls of a single die. Even in a complicated game in which 40 types of space aliens might attack the player, you need only 40 whole numbers generated to satisfy the program requirements.

For example, suppose you need a random number from 1 to 10. To change any value generated by the `Math.random()` method to fall between 0 and 10, you can multiply the generated number by 10. For example, the last three numbers in Figure D-2 would become approximately 2.75, 6.15, and 9.66. Then, you can eliminate the fractional part of each number by casting it to an `int`; after this step, every generated number will be a value from 0 to 9 inclusive. Finally, you can add 1 to a value so it falls in the range from 1 to 10 instead of 0 to 9. In short, the following statement generates a random number from 1 through 10 inclusive, and assigns it to `ran`:

```
int ran = 1 + (int)(Math.random() * 10);
```

Suppose that, instead of 1 through 10, you need random numbers from 1 through 13. (For example, standard decks of playing cards have 13 values from which you might want to select.) When you use the modulus operator (`%`) to find a remainder, the remainder is always a value from 0 to one less than the number. For example, if you divide any number by 4, the remainder is always a value from 0 through 3. Therefore, to find a number from 1 through 13, you can use a statement like the following:

```
int ranCardValue = ((int)(Math.random() * 100) % 13 + 1);
```

In this statement, a randomly generated value (for example, 0.447) is multiplied by 100 (producing 44.7). The result is converted to an `int` (44). The remainder after dividing by 13 is 5. Finally, 1 is added so the result is 1 through 13 instead of 0 through 12 (giving 6). In short, the general format for assigning a random number to a variable is:

```
int result = ((int)(Math.random() * 100) %  
    HIGHEST_VALUE_WANTED + LOWEST_VALUE_WANTED);
```



Instead of using 100 as the multiplier, you might prefer to use a higher value such as 1,000 or 10,000. For most games, the randomness generated using 100 is sufficient.

Using the Random Class

The `Random` class provides a generator that creates a list of random numbers. To use this class, you must use one of the following import statements:

```
import java.util.*;  
import java.util.Random;
```

You also must instantiate a random-number generator object using one of the following constructors:

- `Random()`, in which the seed comes from the operating system. This constructor sets the seed of the random-number generator to a value that is probably distinct from any other invocation of this constructor.
- `Random(long seed)`, in which you provide a starting seed so that your results are reproducible

After you create a random-number generator object, you can use any of the methods in Table D-2 to get the next random number from the generator.

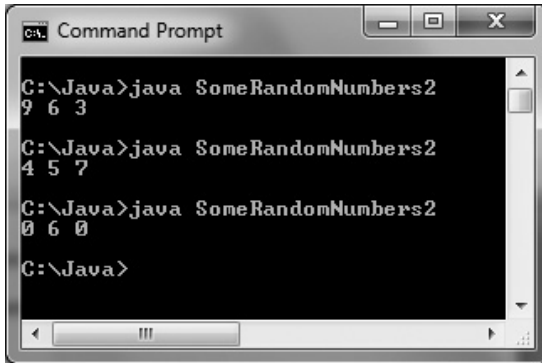
Method	Explanation
<code>nextInt(int n)</code>	Returns a pseudorandom <code>int</code> value between 0 (inclusive) and the specified value <code>n</code> (exclusive), drawn from the random-number generator's sequence
<code>nextInt()</code>	Returns a pseudorandom <code>int</code> value between 0 (inclusive) and 1.0 (exclusive), drawn from the random-number generator's sequence
<code>nextLong()</code>	Returns the next pseudorandom <code>long</code> value from the generator's sequence
<code>nextFloat()</code>	Returns the next pseudorandom <code>float</code> value between 0.0 and 1.0 from the generator's sequence
<code>nextDouble()</code>	Returns the next pseudorandom <code>double</code> value between 0.0 and 1.0 from the generator's sequence
<code>nextBoolean()</code>	Returns the next pseudorandom <code>boolean</code> value from the generator's sequence

Table D-2 Selected Random class methods

For example, Figure D-3 contains an application that declares a `Random` generator named `ran`, using the version of the constructor that takes no arguments. This ensures that the results are different each time the application runs. The program then defines `LIMIT` as 10 and calls `ran.nextInt(LIMIT)` three times, displaying the results (see Figure D-4).

```
import java.util.*;
public class SomeRandomNumbers2
{
    public static void main(String[] args)
    {
        Random ran = new Random();
        final int LIMIT = 10;
        System.out.print(ran.nextInt(LIMIT) + " ");
        System.out.print(ran.nextInt(LIMIT) + " ");
        System.out.println(ran.nextInt(LIMIT));
    }
}
```

Figure D-3 The `SomeRandomNumbers2` class



```
Command Prompt
C:\Java>java SomeRandomNumbers2
9 6 3
C:\Java>java SomeRandomNumbers2
4 5 7
C:\Java>java SomeRandomNumbers2
0 6 0
C:\Java>
```

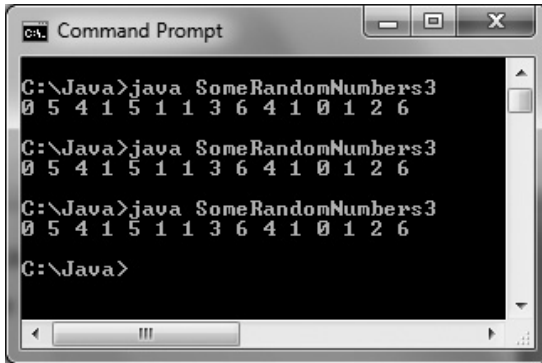
Figure D-4 Three executions of the SomeRandomNumbers2 program

In Figure D-4, each displayed value falls between 0 and LIMIT. (Of course, to select values between 1 and LIMIT inclusive, you could add 1 to each result.)

Figure D-5 shows a class using the version of the Random constructor that takes an argument (shaded). In this example, a value between 0 and 6 inclusive is generated 15 times. Figure D-6 shows the output when the program is run three times. Although the 15 numbers displayed for each execution constitute a random list, the list is identical in each program execution. You use a seed when you want random but reproducible results. For games, you usually want to use the no-argument version of the Random constructor.

```
import java.util.*;
public class SomeRandomNumbers3
{
    public static void main(String[] args)
    {
        Random ran = new Random(129867L);
        final int TIMES = 15;
        final int LIMIT = 7;
        for(int x = 0; x < TIMES; ++x)
            System.out.print(ran.nextInt(LIMIT) + " ");
        System.out.println();
    }
}
```

Figure D-5 The SomeRandomNumbers3 class



```
C:\Java>java SomeRandomNumbers3
0 5 4 1 5 1 1 3 6 4 1 0 1 2 6

C:\Java>java SomeRandomNumbers3
0 5 4 1 5 1 1 3 6 4 1 0 1 2 6

C:\Java>java SomeRandomNumbers3
0 5 4 1 5 1 1 3 6 4 1 0 1 2 6

C:\Java>
```

Figure D-6 Three executions of the SomeRandomNumbers3 program

Key Terms

A **random number** is a number whose value cannot be predicted.

A **seed** is a starting value.

Pseudorandom numbers appear to be random but are the same set of numbers whenever the seed is the same.

