

The Random Class

Random numbers are extremely useful, for example, in generating moves in a game or as test data for computer programs. If one is asked to "pick a number between one and a hundred", the task seems simple enough. But, if you need truly random numbers (each number is equally probable!), and if you want to generate them from a computer, the task is quite tricky as it turns out. Mathematicians have laboured over this problem, and have devised robust techniques to generate random numbers. However, computers are deterministic (all actions are predictable — at some level!), and, so, generating numbers that are "truly" random is not possible. However, we can get pretty close. Algorithms that generate random numbers, admittedly, provide "pseudo-random" numbers. But, for most purposes this is good enough.

Java's `Random` class in the `java.util` package includes a variety of methods to generate pseudo-random numbers. These are summarized in Table 1.

Table 1. Methods of the Random Class

Methods	Description
Constructor	
<code>Random()</code>	creates a new random number generator; returns a reference to the new object
Instance Methods	
<code>nextBoolean()</code>	returns a <code>boolean</code> : the next pseudo-random, uniformly distributed <code>boolean</code> value from this random number generator's sequence
<code>nextDouble()</code>	returns a <code>double</code> : the next pseudo-random, uniformly distributed <code>double</code> value between 0.0 and 1.0 from this random number generator's sequence
<code>nextGaussian()</code>	returns a <code>double</code> : the next pseudo-random, Gaussian ("normally") distributed <code>double</code> value with mean 0.0 and standard deviation 1.0 from this random number generator's sequence
<code>nextInt()</code>	returns an <code>int</code> : the next pseudo-random, uniformly distributed <code>int</code> value from this random number generator's sequence
<code>nextInt(int n)</code>	returns an <code>int</code> : a pseudo-random, uniformly distributed <code>int</code> value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.

Example: Random Bits

Let's put the `Random` class to work in a demo program. The `RandomBits` program in Figure 1 generates a series of random bits (0 or 1). Each bit is represented as a character and placed in a 10-character string buffer (initially containing only spaces) at a random location in the string. After each bit is inserted, the string buffer is printed. A delay of 100 ms between print statements improves the visual output.

```

1  import java.util.*;
2
3  public class RandomBits
4  {
5      public static void main(String[] args)
6      {
7          long t1 = System.currentTimeMillis();
8          int temp = 0;
9          StringBuffer sb = new StringBuffer("          "); // 10 spaces
10         Random r = new Random();
11         while(true)
12         {
13             int t2 = (int)(System.currentTimeMillis() - t1) / 100;
14             if (t2 != temp)
15             {
16                 int i = r.nextInt(10);
17                 String bit = r.nextBoolean() ? "1" : "0";
18                 sb.replace(i, i + 1, bit);
19                 System.out.println(sb);
20                 temp = t2;
21             }
22         }
23     }
24 }

```

Figure 1. RandomBits.java

A sample 10-second run of RandomBits is shown in Figure 2. To conserve space, the 100 lines of output are shown across four columns.

```

0          0011010101  1010011001  1100011010
0 0        0011010100  1010011011  1100011010
0 0 1      0011010100  1010011011  1110011010
0 0 1      0011010100  1010011011  1110011000
1 0 1      0010010100  1010011010  1110111000
1 00 1     1010010100  1010011010  1110111000
1 10 1     1011010100  1010011010  1110111000
1 11 1     1011010101  1010011010  1110111010
0 1 11 1   1011010101  1010011010  1110111010
0 1 011 1  0011010101  1000011010  1111111010
001 011 1  0011010101  1000011010  1111111010
0011011 1  0011010101  1000001010  1111111010
0111011 1  0010010101  1000001010  1101111010
0111011 0  0010010111  1000001011  1100111010
0111011 0  0010011111  1000001011  1100111010
0111011 1  0010011111  1000001011  1100111010
0111011 0  0010011111  1000001010  1110111010
01110111 0  0010011101  1000001010  1110011010
01100111 0  0010011101  1000001010  1010011010
0110011100 0010011111  1000001000  1010001010
0010011100 0010011101  1000001010  0010001010
0010011100 0010011101  1000001010  0010001110
0010010100 1010011101  1100001010  0110001110
0010010100 1010011101  1100001010  0110001110
0011010100 1010011101  1100101010  0110000110
0011010101 1010011001  1100111010  0100000110

```

Figure 2. Sample output from RandomBits.java

This program is little more than pesky fun; however it brings together several methods and classes. The `currentTimeMillis()` method of the `System` class is used in creating the 100 ms delay between successive print statements. The `replace()` method of the `StringBuffer` class is used to place one-character strings ("0" or "1") in a dynamic string. The `nextInt()` and `nextBoolean()` methods of the `Random` class provide random indices and random bits.

The implementation of the 100 ms time delay is very similar to the delay demonstrated earlier in `CountSeconds.java`, so we needn't say more here. The act of placing random characters in a string at random locations in the string requires two random numbers. The string contains ten characters, so the indices vary from 0 to 9. To generate a random index between 0 and 9, the `nextInt()` method is called with "10" as an argument (line 16). A pseudo-random integer between 0 and 9 is returned and assigned to the `int` variable `i`. In line 17, a pseudo-random bit is generated using the `nextBoolean()` method. Since `nextBoolean()` returns a `boolean` (`true` or `false`) the selection operation helps convert the result to the string "0" or the string "1". Recall that the statement

```
String bit = r.nextBoolean() ? "1" : "0";
```

is equivalent to

```
String bit;
if (r.nextBoolean())
    bit = "1";
else
    bit = "0";
```

Note that a relational test is always a test for "true"; so, the expression

```
r.nextBoolean()
```

is equivalent to

```
r.nextBoolean() == true
```

A `StringBuffer` object `sb` is instantiated in line 9 and initialized with ten space characters. One character is replaced with "0" or "1" in line 18, and the new string is printed in line 19. The critical step occurs in line 18 with the `replace()` method. The method replaces strings; so the arguments are the one-character `String` object `bit` and the indices `i` and `i + 1`.

Note in the sample output (Figure 2) that both the value replaced and the position appear random from one line to the next. At most, one character changes. In some cases, no character changes. This occurs approximately half the time — when the new random bit happens to match the existing bit.

The `Random` class documentation in the Java API documentation, includes details on the algorithms for generating random numbers. Consult this for more details or see references [4] or [5].

If all you want is a random double, as provided by the `nextDouble()` method, you can avoid instantiating a `Random` object. The `Math` class of the `java.lang` package includes a method called `random()` with the same effect. Once again, behind-the-scenes activities do the work and hide the details. The statement

```
double x = Math.random();
```

first creates a single new pseudo-random number generator, exactly as if by the expression

```
new java.util.Random()
```

and then retrieves a random double as if by the expression

```
nextDouble()
```

The new pseudo-random number generator is used thereafter for all calls to `random()`.

Example: Random Quilt

The next example is an applet using the `nextDouble()` method of the `Math` class. `RandomQuilt` randomly generates a quilt-like pattern of colored patches in a graphics window. The pattern changes ten times per second. The source code is shown in Figure 3.

```
1 import java.awt.*;
2 import java.applet.*;
3
4 public class RandomQuilt extends Applet
5 {
6     public void paint(Graphics g)
7     {
8         long t1 = System.currentTimeMillis();
9         int temp = 0;
10        while (true)
11        {
12            // updates 10 times per seconds
13            int t2 = (int)(System.currentTimeMillis() - t1) / 100;
14            if (t2 != temp)
15            {
16                // generate arguments as random integers
17                int x = (int)(Math.random() * 1000) % (XSIZE - 1);
18                int y = (int)(Math.random() * 1000) % (YSIZE - 1);
19                int width = (int)(Math.random() * 1000) % (XSIZE - 1 - x);
20                int height = (int)(Math.random() * 1000) % (YSIZE - 1 - y);
21                int red = (int)(Math.random() * 1000) % (256);
22                int green = (int)(Math.random() * 1000) % (256);
23                int blue = (int)(Math.random() * 1000) % (256);
24
25                // set random color
26                g.setColor(new Color(red, green, blue));
27
28                // draw filled rectangle: random position, random proportions
29                g.fillRect(x, y, width, height);
30
31                temp = t2;
32            }
33        }
34    }
35    static final int XSIZE = 250;
36    static final int YSIZE = 150;
37 }
```

Figure 3. `RandomQuilt.java`

A screen snapshot of this applet after about 10 seconds of execution is shown in Figure 4. Of course, it looks much better in color. Note that the pattern of patches continually changes, with a new patch appearing every 100 ms (ten times per second).

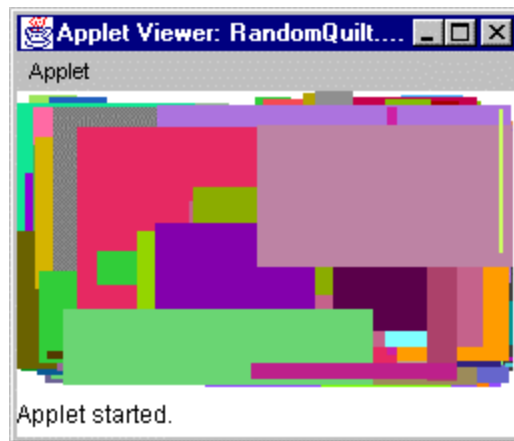


Figure 4. Output of RandomQuilt applet

The program executes in an infinite loop and must be terminated by entering Control-c on the keyboard from the DOS window, or by closing the applet window. The `currentTimeMillis()` method of the `System` class is used in the same manner as in the `CountSeconds` and `RandBits` programs discussed earlier.

Within the main loop, seven random numbers are generated for each new rectangle (lines 17-23). Four numbers specify the *x-y* location and *width* and *height* of the rectangle, and three numbers specify the *red*, *green*, and *blue* components of the rectangle's color. Since the `random()` method returns a `double` between 0.0 and 1.0, the result is multiplied by 1000, then cast to an `int`. The result is a random integer between 0 and 1000. This is scaled to an appropriate range using the `mod (%)` operator. The "appropriate range" in the case of the rectangle parameters is such that the rectangle fully lies within the graphics window. The appropriate range for the color parameters is 0-255.

With these seven random numbers, the drawing color is set in line 26 and a filled rectangle is drawn in line 29. The `setColor()` and `fillRect()` methods are in the `Graphics` class in the `java.awt` package (see the Java API documentation).