

The System Class

We have already met the all-important data fields defined in the `System` class to setup the standard input and standard output streams. The `System` class also includes a variety of special-purpose methods (see the Java API documentation); however, we will only discuss one here. The `currentTimeMillis()` method returns a `long` equal to the current time in milliseconds (ms). Note that 1 ms equals 10^{-3} seconds. The value returned is relative to the beginning of January 1, 1970; so, in most practical applications, it is the difference between two successive calls to `currentTimeMillis()` that is of interest. Java includes a comprehensive set of classes for dates and times, and we'll explore these later. For the moment, let's see the possibilities for the `currentTimeMillis()` method in the `System` class. A simple "seconds counter" is demonstrated in the program `CountSeconds` (see Figure 1).

```
1 public class CountSeconds
2 {
3     public static void main(String[] args)
4     {
5         long start = System.currentTimeMillis();
6         long seconds = 0;
7         while(true)
8         {
9             long temp = System.currentTimeMillis();
10            long newSeconds = (temp - start) / 1000;
11            if (newSeconds != seconds)
12            {
13                System.out.print(newSeconds + " ");
14                seconds = newSeconds;
15            }
16        }
17    }
18 }
```

Figure 1. `CountSeconds.java`

This program generates the following output:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 etc.
```

Each number is printed "precisely" one second following the previous one. The timing of the output is as precise as the host system's clock.

The main loop (lines 7-16) is setup with `while (true)`, so it is an infinite loop. The program executes indefinitely and continues to count seconds until the user enters Control-c to terminate the program. The `currentTimeMillis()` method is first called in line 5. The value read is a very large integer since it equals the number of milliseconds since January 1, 1970. It's not much use, as is. Nevertheless, it is assigned to the `long` variable `start`. In the next line, a `long` variable `seconds` is declared and initialized with zero.

Within the `while` loop, `currentTimeMillis()` is called again (line 9), with the new value assigned to the `long` variable `temp` (line 9). The original value in `start` is subtracted from `temp` and the result is divided by 1000. This result, which equals the elapsed time in seconds between the two calls, is assigned to the `long` variable `newSeconds` (line 10). (Note: the result of the expression `(temp - start) / 100` yields a `long` integer.)

All the tricky stuff is in the `if` statement in lines 11-15. First, the long variables `newSeconds` and `seconds` are compared. If they are not equal, the statement block is skipped and the while loop begins again with a new call to `currentTimeMillis()`. Only after 1000 ms (1 second) has elapsed between the first two calls to `currentTimeMillis()` will `newSeconds` finally equal 1. At this point, the relational test in the `if` statement passes and the statement block is executed (lines 12-15). The value of `newSeconds` is printed (It equals "1" the first time through!), and then `seconds` is reassigned to `newSeconds` (line 14). This last step is crucial, as it ensures the relational expression in the `if` statement fails again, repeatedly, until another 1000 ms has elapsed.

The while loop in `CountSeconds` executes "many" times for each value printed. The actual number varies on the speed of the host system, and on the number and activity of background processes. To investigate this, `CountSeconds` was re-worked with a few modifications (see Figure 2).

```
1 public class CountSeconds2
2 {
3     public static void main(String[] args)
4     {
5         long start = System.currentTimeMillis();
6         long seconds = 0;
7         int i = 0;
8         while(true)
9         {
10            i++;
11            long temp = System.currentTimeMillis();
12            long newSeconds = (temp - start) / 1000;
13            if (newSeconds != seconds)
14            {
15                System.out.println(i);
16                i = 0;
17                seconds = newSeconds;
18            }
19        }
20    }
21 }
```

Figure 2. `CountSeconds2.java`

`CountSeconds2` leaves in tact most of the activities in `CountSeconds`. However, a new variable is introduced. The `int` variable `i` is declared and initialized to zero in line 7. Each time through the while loop, `i` is incremented (line 10); however, it is only printed once per second, inside the `if` statement (line 15). After `i` is printed, it is reset to zero (line 16). A sample ten-second run of this program is shown below.

6071
4744
4173
5459
5470
5762
5450
5476
5463
5411

So, the `while` loop executes approximately 5400 times each second. The high count was 6071, the low count was 4173. The variation is due to background processing on the host system. This output was generated from a notebook computer with a 166 MHz Pentium MMX processor running the *Windows98* operating system. The counts above will differ for machines with faster or slower CPU clock speeds.