Trail: Essential Java Classes
Lesson: Threads: Doing Two or More Tasks at Once

Using the Timer and TimerTask Classes

This section discusses practical aspects of using timers to schedule tasks. The `Timer` class in the `java.util` package schedules instances of a class called `TimerTask`. `Reminder.java` is an example of using a timer to perform a task after a delay.

```
import java.util.Timer;
import java.util.TimerTask;

/**
 * Simple demo that uses java.util.Timer to schedule a task
 * to execute once 5 seconds have passed.
 */

public class Reminder {
    Timer timer;

    public Reminder(int seconds) {
        timer = new Timer();
        timer.schedule(new RemindTask(), seconds*1000);
        }

    class RemindTask extends TimerTask {
        public void run() {
            System.out.format("Time's up!%n");
            timer.cancel(); //Terminate the timer thread
```

```
            }
        }

        public static void main(String args[]) {
            new Reminder(5);
            System.out.format("Task scheduled.%n");
        }
    }
```

When you run the example, you see this first.

```
Task scheduled.
```

Five seconds later, you see this.

```
Time's up!
```

This simple program illustrates the following basic parts of implementing and scheduling a task to be executed by a timer thread:

- Implement a custom subclass of `TimerTask`. The `run` method contains the code that performs the task. In this example, the subclass is named `RemindTask`.
- Create a thread by instantiating the `Timer` class.
- Instantiate the `TimerTask` object (`new RemindTask()`).
- Schedule the timer task for execution.

This example uses the `schedule` method, with the timer task as the first argument and the delay in milliseconds (`5000`) as the second argument. Another way of scheduling a task is to specify the time when the task should execute. For example, the following code schedules a task for execution at 11:01 P.M.

```
//Get the Date corresponding to 11:01:00 pm today.
Calendar calendar = Calendar.getInstance();
calendar.set(Calendar.HOUR_OF_DAY, 23);
```

```
        calendar.set(Calendar.MINUTE, 1);
        calendar.set(Calendar.SECOND, 0);
        Date time = calendar.getTime();

        timer = new Timer();
        timer.schedule(new RemindTask(), time);
```

Stopping Timer Threads

By default, a program keeps running as long as its timer threads are running. You can terminate a timer thread in four ways:

- Invoke `cancel` on the timer. You can do this from anywhere in the program, such as from a timer task's `run` method.
- Make the timer's thread a "daemon" by creating the timer like this: `new Timer(true)`. If the only threads left in the program are daemon threads, the program exits.
- After all the timer's scheduled tasks have finished executing, remove all references to the `Timer` object. Eventually, the timer's thread will terminate.
- Invoke the `System.exit` method, which makes the entire program (and all its threads) exit.

The `Reminder` example uses the first scheme, invoking the `cancel` method from the timer task's `run` method. Making the timer thread a daemon wouldn't work because the program needs to keep running until the timer's task executes.

Sometimes, timer threads aren't the only threads that can prevent a program from exiting when expected. For example, if you use the AWT at all (even if only to make beeps), it automatically creates a nondaemon thread that keeps the program alive. The following modification of `Reminder` adds beeping, which requires us to also add a call to the `System.exit` method to make the program exit. Significant changes are in boldface. You can find the source code in `ReminderBeep.java`.

```
    public class ReminderBeep {
        ...
```

```
        public ReminderBeep(int seconds) {
            toolkit = Toolkit.getDefaultToolkit();
            timer = new Timer();
            timer.schedule(new RemindTask(), seconds*1000);
        }

        class RemindTask extends TimerTask {
            public void run() {
                System.out.format("Time's up!%n");
                toolkit.beep();
                //timer.cancel(); //Not necessary because
                                  //we call System.exit.
                System.exit(0);   //Stops the AWT thread
                                  //(and everything else).
            }
        }
        ...
    }
```

Performing a Task Repeatedly

The following is an example of using a timer to perform a task once per second.

```
    public class AnnoyingBeep {
        Toolkit toolkit;
        Timer timer;

        public AnnoyingBeep() {
            toolkit = Toolkit.getDefaultToolkit();
            timer = new Timer();
            timer.schedule(new RemindTask(),
                           0,        //initial delay
                           1*1000);  //subsequent rate
        }

        class RemindTask extends TimerTask {
```

```
            int numWarningBeeps = 3;
            public void run() {
                if (numWarningBeeps > 0) {
                    toolkit.beep();
                    System.out.format("Beep!%n");
                    numWarningBeeps--;
                } else {
                    toolkit.beep();
                    System.out.format("Time's up!%n");
                    //timer.cancel(); //Not necessary because
                                      //we call System.exit
                    System.exit(0);   //Stops the AWT thread
                                      //(and everything else)
                }
            }
        }
        ...
    }
```

You can find the entire program in AnnoyingBeep.java. When you execute it, you see the following output (comments about timing are shown in italics).

```
Task scheduled.
Beep!
Beep!      //one second after the first beep
Beep!      //one second after the second beep
Time's up! //one second after the third beep
```

The `AnnoyingBeep` program uses a three-argument version of the `schedule` method to specify that its task should execute once a second, beginning immediately. The following are all the `Timer` methods you can use to schedule repeated executions of tasks:

- `schedule(TimerTask` *`task`*`, long` *`delay`*`, long` *`period`*`)`
- `schedule(TimerTask` *`task`*`, Date` *`time`*`, long` *`period`*`)`
- `scheduleAtFixedRate(TimerTask` *`task`*`, long` *`delay`*`, long` *`period`*`)`

- `scheduleAtFixedRate(TimerTask task, Date firstTime, long period)`

When scheduling a task for repeated execution, use one of the `schedule` methods when smoothness is important and a `scheduleAtFixedRate` method when time synchronization is more important. For example, the `AnnoyingBeep` program uses the `schedule` method, which means that the annoying beeps will all be at least one second apart. If one beep is late for any reason, all subsequent beeps will be delayed. If you decide that the `AnnoyingBeep` program should exit exactly three seconds after the first beep, even if that means two beeps might occur close together if a beep is delayed for any reason, use the `scheduleAtFixedRate` method instead.

More Information About Timers

The timer tasks shown here have been very simple. They do almost nothing and refer only to data that either can be safely accessed from multiple threads or is private to the timer task. As long as your timer task uses only APIs designed to be thread-safe, such as the methods in the `Timer` class, implementing timers is relatively straightforward. However, if your timer implementation depends on shared resources, such as data used by other places in your program, you need to be careful. To find out more about this, see the Synchronizing Threads section.