23.3 Creating and Executing Threads with the `Executor` Framework

This section demonstrates how to perform concurrent tasks in an application by using `Executors` and `Runnable` objects.

Creating Concurrent Tasks with the `Runnable` Interface

You implement the `Runnable` interface (of package `java.lang`) to specify a task that can execute concurrently with other tasks. The `Runnable` interface declares the single method `run`, which contains the code that defines the task that a `Runnable` object should perform.

Executing `Runnable` Objects with an `Executor`

To allow a `Runnable` to perform its task, you must execute

it. An `Executor` object executes `Runnable`s. It does this by creating and managing a group of threads called a **thread pool**. When an `Executor` begins executing a `Runnable`, the `Executor` calls the `Runnable` object's `run` method.

The `Executor` interface declares a single method named `execute` which accepts a `Runnable` as an argument. The `Executor` assigns every `Runnable` passed to its `execute` method to one of the available threads in the thread pool. If there are no available threads, the `Executor` creates a new thread or waits for a thread to become available and assigns that thread the `Runnable` that was passed to method `execute`.

Using an `Executor` has many advantages over creating threads yourself. `Executor`s can *reuse existing threads* to eliminate the overhead of creating a new thread for each task and can improve performance by *optimizing the number of threads* to ensure that the processor stays busy, without creating so many threads that the application runs out of resources.

Software Engineering Observation 23.2

Though it's possible to create threads explicitly, it's recommended that you use the `Executor` *interface to manage the execution of* `Runnable` *objects.*

Using Class `Executors` to Obtain an `ExecutorService`

The `ExecutorService` **interface** (of package `java.util.concurrent`) *extends* `Executor` and declares various methods for managing the life cycle of an `Executor`. You obtain an `ExecutorService` object by calling one of the `static` methods declared in class `Executors` (of package `java.util.concurrent`). We use interface `ExecutorService` and a method of class `Executors` in our example (Fig. 23.4), which executes three tasks.

Implementing the `Runnable` Interface

Class `PrintTask` (Fig. 23.3) implements `Runnable` (line 5), *so that multiple* `PrintTask`*s can execute concurrently*. Variable `sleepTime` (line 7) stores a random integer value from 0 to 5 seconds created in the `PrintTask` constructor (line 15). Each thread running a `PrintTask` sleeps for the amount of time specified by `sleepTime`, then outputs its task's name and a message indicating that it's done sleeping.

```
1   // Fig. 23.3: PrintTask.java
```

```
2   // PrintTask class sleeps for a random time from
3   import java.security.SecureRandom;
4
5   public class PrintTask implements Runnable {
6      private static final SecureRandom generator =
7      private final int sleepTime; // random sleep
8      private final String taskName;
9
10     // constructor
11     public PrintTask(String taskName) {
12        this.taskName = taskName;
13
14        // pick random sleep time between 0 and 5
15        sleepTime = generator.nextInt(5000); // mi
16     }
17
18     // method run contains the code that a thread
19     @Override
20     public void run() {
21        try { // put thread to sleep for sleepTime
22           System.out.printf("%s going to sleep fo
23              taskName, sleepTime);
24           Thread.sleep(sleepTime); // put thread
25        }
26        catch (InterruptedException exception) {
27           exception.printStackTrace();
28           Thread.currentThread().interrupt(); //
29        }
30
31        // print task name
32        System.out.printf("%s done sleeping%n", ta
33     }
34  }
```

Fig. 23.3

`PrintTask` class sleeps for a random time from 0 to 5 seconds.

A `PrintTask` executes when a thread calls the `PrintTask`'s `run` method. Lines 22–23 display a message indicating the currently executing task's name and that the task is going to sleep for `sleepTime` milliseconds. Line 24 invokes `static` `Thread` method `sleep` to place the thread in the *timed waiting* state for the specified amount of time. At this point, the thread loses the processor, and the system allows another thread to execute. When the thread awakens, it reenters the *runnable* state. When the `PrintTask` is assigned to a processor again, line 32 outputs a message indicating that the task is done sleeping, then method `run` terminates. The `catch` at lines 26–29 is required because method `sleep` might throw a *checked* `InterruptedException` if a sleeping thread's `interrupt` method is called.

Let the Thread Handle `InterruptedException`s

It's considered good practice to let the executing thread handle `InterruptedException`s. Normally, you'd do this by declaring that method `run` throws the exception, rather than catching the exception. However, recall from Chapter 11 that when you override a method, the `throws` clause may contain only the same or a subset of the exception types declared in the original method's `throws` clause. `Runnable` method

`run` does not have a `throws` clause, so we cannot provide one in line 20. To ensure that the executing thread receives the `InterruptedException`, line 28 first obtains a reference to the currently executing `Thread` by calling `static` method `currentThread`, then uses that `Thread`'s `interrupt` method to deliver the `InterruptedException` to the current thread.1

1. For detailed information on handling thread interruptions, see Chapter 7 of *Java Concurrency in Practice* by Brian Goetz, et al., Addison-Wesley Professional, 2006.

Using the `ExecutorService` to Manage Threads That Execute `PrintTasks`

Figure 23.4 uses an `ExecutorService` object to manage threads that execute `PrintTasks` (as defined in Fig. 23.3). Lines 9–11 in Fig. 23.4 create and name three `PrintTask`s to execute. Line 16 uses `Executors` method `newCachedThreadPool` to obtain an `ExecutorService` that creates new threads if no existing threads are available to reuse. These threads are used by the `ExecutorService` to execute the `Runnable`s.

```
1   // Fig. 23.4: TaskExecutor.java
2   // Using an ExecutorService to execute Runnables
3   import java.util.concurrent.Executors;
```

```
import java.util.concurrent.ExecutorService;

public class TaskExecutor {
   public static void main(String[] args) {
      // create and name each runnable
      PrintTask task1 = new PrintTask("task1");
      PrintTask task2 = new PrintTask("task2");
      PrintTask task3 = new PrintTask("task3");

      System.out.println("Starting Executor");

      // create ExecutorService to manage thread
      ExecutorService executorService = Executor

      // start the three PrintTasks
      executorService.execute(task1); // start t
      executorService.execute(task2); // start t
      executorService.execute(task3); // start t

      // shut down ExecutorService--it decides w
      executorService.shutdown();

      System.out.printf("Tasks started, main end
   }
}
```

```
Starting Executor
Tasks started, main ends

task1 going to sleep for 4806 milliseconds
task2 going to sleep for 2513 milliseconds
task3 going to sleep for 1132 milliseconds
task3 done sleeping
task2 done sleeping
task1 done sleeping
```

```
              Starting Executor
task1 going to sleep for 3161 milliseconds.
 task3 going to sleep for 532 milliseconds.
task2 going to sleep for 3440 milliseconds.
         Tasks started, main ends.

            task3 done sleeping
            task1 done sleeping
            task2 done sleeping
```

Fig. 23.4

Using an `ExecutorService` to execute `Runnable`s.

Lines 19–21 each invoke the `ExecutorService`'s `execute` method, which executes its `Runnable` argument (in this case a `PrintTask`) at some time in the future. The specified task may execute in one of the threads in the `ExecutorService`'s thread pool, in a new thread created to execute it, or in the thread that called the `execute` method—the `ExecutorService` manages these details. Method `execute` returns immediately from each invocation—the program does *not* wait for each `PrintTask` to finish. Line 24 calls `ExecutorService` method `shutdown`, which prevents the `ExecutorService` from accepting new tasks, but *continues executing tasks that have already been submitted*. Once all of the previously submitted tasks have completed, the `ExecutorService` terminates. Line 26 outputs a message indicating that the tasks were started and

the `main` thread is finishing its execution.

Main Thread

The code in `main` executes in the **main thread**, which is created by the JVM. The code in the `run` method of `PrintTask` (lines 19–33 of Fig. 23.3) executes whenever the `Executor` starts each `PrintTask`—again, sometime after they're passed to the `ExecutorService`'s `execute` method (Fig. 23.4, lines 19–21). When `main` terminates, the program itself continues running until the submitted tasks complete.

Sample Outputs

The sample outputs show each task's name and sleep time as the thread goes to sleep. The thread with the shortest sleep time *in most cases* awakens first, indicates that it's done sleeping and terminates. In Section 23.8, we discuss multithreading issues that could prevent the thread with the shortest sleep time from awakening first. In the first output, the `main` thread terminates *before* any of the `PrintTask`s output their names and sleep times. This shows that the `main` thread runs to completion before any of the `PrintTask`s gets a chance to run. In the second output, all of the `PrintTask`s output their names and sleep times *before* the `main` thread terminates. This shows that the `PrintTask`s started executing before the main thread terminated. Also, notice in

the second example output, `task3` goes to sleep before `task2`, even though we passed `task2` to the `ExecutorService`'s `execute` method before `task3`. This illustrates the fact that *we cannot predict the order in which the tasks will start executing, even if we know the order in which they were created and started*.

Waiting for Previously Scheduled Tasks to Terminate

After scheduling tasks to execute, you'll typically want to *wait for the tasks to complete*—for example, so that you can use the tasks' results. After calling method `shutdown`, you can call `ExecutorService` method `awaitTermination` to wait for scheduled tasks to complete. We demonstrate this in Fig. 23.7. We purposely did not call `awaitTermination` in Fig. 23.4 to demonstrate that a program can continue executing after the main thread terminates.