IN THIS CHAPTER

- Examining threads
- Understanding thread safety and how to ensure it
- Creating threads from the Thread class
- Working with the Runnable interface
- Creating threads that cooperate
- Interrupting threads
- Working with Virtual Threads

COPYRIGHTED MATERIAL

Chapter 1
Programming Threads

Have you ever seen a plate-spinning act, in which a performer spins plates or bowls on top of poles, keeping multiple plates spinning at the same time, running from pole to pole to give each plate a little nudge — just enough to keep it going? The world record is 108 plates kept simultaneously spinning.

In Java, *threads* are the equivalent of plate spinning. Threads let you divide the work of an application into separate pieces, all of which then run simultaneously. The result is often a faster and more efficient program, but along with the increased speed come more difficult programming and debugging.

Truthfully, the subtleties of threaded programming are a topic for computer science majors, but the basics of working with threads aren't all that difficult to understand. In this chapter, I focus on those basics and leave the advanced techniques for the grad students.



WARNING

The main application I use to illustrate threading in this chapter simulates the countdown clock for the launch of a spacecraft. Working with threads isn't really rocket science, but threading is used to solve difficult programming problems. You invariably find yourself trying to get two or more separate pieces of code to coordinate their activities, and that's not as easy as you might think at first guess. As a result, I can't possibly talk about threading without getting into some challenging mental exercises, so be prepared to spend some mental energy figuring out how it works.



TIP

The listings in this chapter, as well as throughout the book, are available at `www.dummies.com/go/javaaiofd8e`.

Understanding Threads

A *thread* is a single sequence of executable code within a larger program. All the programs shown so far in this book have used just one thread — the *main thread* that starts automatically when you run the program — but Java lets you create programs that start additional threads to perform specific tasks.

You're probably familiar with programs that use threads to perform several tasks at the same time. Here are some common examples:

- Web browsers can download files while letting you view web pages. When you download a file in a web browser, the browser starts a separate thread to handle the download.
- Email programs don't make you wait for all your messages to download before you can read the first message. Instead, these programs use separate threads to display and download messages.
- Word processors can print long documents in the background while you continue to work. These programs start a separate thread to handle print jobs.
- Word processors can also check your spelling as you type. Depending on how the word processor is written, it may run the spell check in a separate thread.
- Game programs commonly use several threads to handle different parts of the game to improve overall responsiveness.
- All GUI-based programs use at least two threads — one thread to run the application's main logic and another thread to monitor mouse and keyboard events. You find out about creating GUI programs in Java in Book 6.



TIP

» Indeed, the Java Virtual Machine (JVM) itself uses threading for some of its housekeeping chores. The garbage collector, for example, runs as a separate thread so it can constantly monitor the state of the JVM's memory and decide when it needs to create some free memory by removing objects that are no longer being used.

Creating a Thread

Suppose you're developing software for NASA, and you're in charge of the program that controls the final 20 seconds of the countdown for a manned spacecraft. Your software has to coordinate several key events that occur when the clock reaches certain points:

» **T minus 16 seconds:** Flood launch pad. This event releases 350,000 gallons of water onto the launch pad, which helps protect the spacecraft systems during launch. (Nearly all of the huge white plume of "smoke" you see in the early moments of a rocket launch is not actually smoke, but steam that results when the intense heat of the rocket exhaust hits this water. And no, the purpose of the water is not to cool the engines or the launch pad. It's to absorb the enormous shock wave created by the massive engines so they don't bounce back and destroy the rocket.)

» **T minus 6 seconds:** Start the main engines. Huge clamps hold the spacecraft in place while the engines build up thrust.

» **T minus 0:** Liftoff! The clamps are released, and the spacecraft flies into space.

For this program, I don't actually start any rocket engines or release huge amounts of water. Instead, I just display messages on the console to simulate these events. But I do create four separate threads to make everything work. One thread manages the countdown clock. The other three threads fire off their respective events at T minus 16 seconds (flood the pad), T minus 6 seconds (fire the engines), and T minus 0 (launch).

For the first attempt at this program, I just get the countdown clock up and running. The countdown clock is represented by a class named `CountDownClock`. All this class does is count down from 20 to 0 at 1-second intervals, displaying messages such as `T minus 20` on the console as it counts. This version of the program doesn't do much of anything, but it does demonstrate how to get a thread going. We'll start by looking at the `Thread` class.

Understanding the Thread class

The `Thread` class lets you create an object that can be run as a thread in a multithreaded Java application. The `Thread` class has quite a few constructors and methods, but for most applications, you need to use only the ones listed in Table 1-1. (Note that this table is here to give you an overview of the `Thread` class and to serve as a reference. Don't worry about the details of each constructor and method just yet. By the end of this chapter, I explain each of the constructors and methods.)

TABLE 1-1 Constructors and Methods of the Thread Class

Constructor	Explanation
`Thread()`	Creates an instance of the `Thread` class. This constructor is the basic `Thread` constructor without parameters.
`Thread(String name)`	Creates a `Thread` object and assigns the specified name to the thread.
`Thread(Runnable target)`	Turns any object that implements an API interface called `Runnable` into a thread.
`Thread(Runnable target, String name)`	Creates a thread from any object that implements `Runnable` and assigns the specified name to the thread.
`static int activeCount()`	Returns the number of active threads.
`static int enumerate(Thread[] t)`	Fills the specified array with a copy of each active thread. The return value is the number of threads added to the array.
`String getName()`	Returns the name of the thread.
`int getPriority()`	Returns the thread's priority.
`void interrupt()`	Interrupts this thread.
`boolean isInterrupted()`	Checks whether the thread has been interrupted.
`void setPriority(int priority)`	Sets the thread's priority.
`void setName(String name)`	Sets the thread's name.
`static void sleep (int milliseconds)`	Causes the currently executing thread to sleep for the specified number of milliseconds.
`void run()`	Is called when the thread is started. Place the code that you want the thread to execute inside this method.
`void start()`	Starts the thread.
`static void yield()`	Causes the currently executing thread to yield to other threads that are waiting to execute.

Extending the Thread class

The easiest way to create a thread is to write a class that extends the `Thread` class. Then all you have to do to start a thread is create an instance of your thread class and call its `start` method.

Listing 1-1 is a version of the `CountDownClock` class that extends the `Thread` class.

LISTING 1-1: **The CountDownClock Class (Version 1)**

```
public class CountDownClock extends Thread                          →1
{
    public void run()                                               →3
    {
        for (int t = 20; t >= 0; t--)                               →5
        {
            System.out.println("T minus " + t);
            try
            {
                Thread.sleep(1000);                                 →10
            }
            catch (InterruptedException e)
            {}
        }
    }
}
```

Here are a few key points to notice in this class:

- **→1:** The `CountDownClock` class extends the `Thread` class. `Thread` is defined in the `java.lang` package, which is imported automatically so you don't have to provide an `import` statement for it.
- **→3:** The `CountDownClock` class has a single method, named `run`. This method is called by Java when the clock thread has been started. All the processing done by the thread must either be in the `run` method or in some other method called by the `run` method.
- **→5:** The `run` method includes a `for` loop that counts down from 20 to 0.
- **→10:** The `CountDownClock` class uses the `sleep` method to pause for 1 second. Because the `sleep` method throws `InterruptedException`, a `try–catch` statement handles this exception. When the exception is caught, it is simply ignored.



REMEMBER

At some point in its execution, the `run` method should either call `sleep` or `yield` to give other threads a chance to execute.



TIP

In modern Java code, you rarely see a class extend the `Thread` class as shown in Listing 1-1. It's almost always better to implement the `Runnable` interface rather than extend the `Thread` class. You learn how to do that in the section "Implementing the Runnable Interface," later in this chapter.

Creating and starting a thread

After you define a class that defines a `Thread` object, you can create and start the thread. Here's the main class for the first version of the countdown application:

```
public class CountDownApp
{
    public static void main(String[] args)
    {
        Thread clock = new CountDownClock();
        clock.start();
    }
}
```

Here a variable of type `Thread` is declared, and an instance of the `CountDownClock` is created and assigned to it. This creates a `Thread` object, but the thread doesn't begin executing until you call its `start` method. (Note that when you call the `start` method, the main program continues to run in its own thread; the new thread runs concurrently with the program's main thread.)

When you run this program, the thread starts counting down in 1-second increments, displaying messages such as the following on the console:

```
T minus 20
T minus 19
T minus 18
.
.
.
T minus 2
T minus 1
T minus 0
```

So far, so good.

Implementing the Runnable Interface

For the threads that trigger specific countdown events such as flooding the launch pad, starting the events, and lifting off, I create another class called `LaunchEvent`. This class uses another technique for creating and starting threads — one that requires a few more lines of code but is more flexible.

The problem with creating a class that extends the `Thread` class is that a class can have only one superclass. What if you'd rather have your thread object extend some other class? In that case, you can create a class that implements the `Runnable` interface rather than extends the `Thread` class. The `Runnable` interface marks an object that can be run as a thread. It has only one method, `run`, that contains the code that's executed in the thread. (The `Thread` class itself implements `Runnable`, which is why the `Thread` class has a `run` method.)

Using the Runnable interface

To use the `Runnable` interface to create and start a thread, you have to do the following:

1. **Create a class that implements `Runnable`.**
2. **Provide a `run` method in the class you created in Step 1.**
3. **Create an instance of the `Thread` class and pass your `Runnable` object to its constructor as a parameter.**

   A `Thread` object is created that can run your `Runnable` class.
4. **Call the `Thread` object's `start` method.**

   The `run` method of your `Runnable` object is called and executes in a separate thread.

The first two of these steps are easy. The trick is in the third and fourth steps, because you can complete them in several ways. Here's one way, assuming that your `Runnable` class is named `RunnableClass`:

```
RunnableClass rc = new RunnableClass();
Thread t = new Thread(rc);
t.start();
```

Java programmers like to be as concise as possible, so you often see this code compressed to something more like

```
Thread t = new Thread(new RunnableClass());
t.start();
```

or even just this:

```
new Thread(new RunnableClass()).start();
```

This single-line version works — provided that you don't need to access the thread object later in the program.

Creating a class that implements Runnable

To sequence the launch events for the rocket launch application, I create a `Runnable` object named `LaunchEvent`. The constructor for this class accepts two parameters: the countdown time at which the event fires and the message that is displayed when the time arrives. The `run` method for this class uses `Thread.sleep` to wait until the desired time arrives. Then it displays the message.

Listing 1-2 shows the code for this class.

LISTING 1-2: **The LaunchEvent Class (Version 1)**

```
public class LaunchEvent implements Runnable                          →1
{
    private int start;
    private String message;

    public LaunchEvent(int start, String message)                     →6
    {
        this.start = start;
        this.message = message;
    }

    public void run()
    {
        try
        {
            Thread.sleep(20000 - (start * 1000));                     →16
        }
        catch (InterruptedException e)
        {}
        System.out.println(message);                                  →20
    }
}
```

The following paragraphs draw your attention to the listing's key lines:

- **→1:** This class implements the `Runnable` interface.
- **→6:** The constructor accepts two parameters: an integer representing the start time (in seconds) and a string message that's displayed when the time arrives. The constructor simply stores these parameter values in private fields.
- **→16:** In the `run` method, the `Thread.sleep` method is called to put the thread to sleep until the desired countdown time arrives. The length of time that the thread should sleep is calculated by the expression `20000 - (start * 1000)`. The countdown clock starts at 20 seconds, which is 20,000 milliseconds. This expression simply subtracts the number of milliseconds that corresponds to the desired start time from 20,000. Thus, if the desired start time is launch minus 6 seconds, the `sleep` method sleeps for 14,000 milliseconds — that is, 14 seconds.
- **→20:** When the thread wakes up, it displays the message passed via its constructor on the console.

Using the CountDownApp class

Now that you've seen the code for the `LaunchEvent` and `CountDownClock` classes, Listing 1-3 shows the code for a `CountDownApp` class that uses these classes to launch a spacecraft.

LISTING 1-3: **The CountDownApp Class (Version 2)**

```
public class CountDownApp
{
    public static void main(String[] args)
    {
        Thread clock = new CountDownClock();                                    →5

        Runnable flood, ignition, liftoff;                                      →7
        flood = new LaunchEvent(16, "Flood the pad!");
        ignition = new LaunchEvent(6, "Start engines!");
        liftoff = new LaunchEvent(0, "Liftoff!");

        clock.start();                                                          →12

        new Thread(flood).start();                                              →14
        new Thread(ignition).start();
        new Thread(liftoff).start();
   }
}
```

The following paragraphs summarize how this program works:

- **→5:** The `main` method starts by creating an instance of the `CountDownClock` class and saving it in the `clock` variable.
- **→7:** Next, it creates three `LaunchEvent` objects to flood the pad at 16 seconds, start the engines at 6 seconds, and liftoff at 0 seconds. These objects are assigned to variables of type `Runnable` named `flood`, `ignition`, and `liftoff`.
- **→12:** The `clock` thread is started. The countdown starts ticking.
- **→14:** Finally, the program starts the three `LaunchEvent` objects as threads. It does this by creating a new instance of the `Thread` class, passing the `LaunchEvent` objects as parameters to the `Thread` constructor, and then calling the `start` method to start the thread. Note that because this program doesn't need to do anything with these threads after they're started, it doesn't bother creating variables for them.

When you run this program, output similar to the following is displayed on the console:

```
T minus 20
T minus 19
T minus 18
T minus 17
T minus 16
Flood the pad!
T minus 15
T minus 14
T minus 13
T minus 12
T minus 11
T minus 10
T minus 9
T minus 8
T minus 7
T minus 6
Start engines!
T minus 5
T minus 4
T minus 3
T minus 2
T minus 1
Liftoff!
T minus 0
```

As you can see, the `LaunchEvent` messages are interspersed with the `CountDownClock` messages. Thus, the launch events are triggered at the correct times.

Note that the exact order in which some of the messages appear may vary slightly. For example, `"Flood the pad!"` might sometimes come before `"T minus 16"` because of slight variations in the precise timing of these independently operating threads. Later in this chapter, the section "Creating Threads That Work Together" shows you how to avoid such inconsistencies.



TIP

You can improve the `main` method for this class by using an `ArrayList` to store the `Runnable` objects. Then you can start all the `LaunchEvent` threads by using an enhanced `for` loop. Here's what the improved code looks like:

```
public static void main(String[] args)
{
    Thread clock = new CountDownClock();
    ArrayList<Runnable> events
        = new ArrayList<Runnable>();
    events.add(new LaunchEvent(16, "Flood the pad!"));
    events.add(new LaunchEvent(6, "Start engines!"));
    events.add(new LaunchEvent(0, "Liftoff!"));
    clock.start();
    for (Runnable e : events)
        new Thread(e).start();
}
```

The advantage of this technique is that you don't need to create a separate variable for each `LaunchEvent`. (Don't forget to add an `import` statement for the `java.util.*` to gain access to the `ArrayList` class.)

Creating Threads That Work Together

Unfortunately, the countdown application presented in the preceding section has a major deficiency: The `CountDownClock` and `LaunchEvent` threads depend strictly on timing to coordinate their activities. After these threads start, they run independently of one another. As a result, random variations in their timings can cause the thread behaviors to change. If you run the program several times in a row, you'll discover that sometimes the `Start engines!` message appears after the `T minus 6` message, and sometimes it appears *before* the `T minus 6` message. That might not seem like a big deal to you, but it probably would be disastrous for the astronauts on the spacecraft. What these classes really need is a way to communicate.

Listing 1-4 shows an improved version of the countdown application that incorporates several enhancements. The `CountDownClock` class in this version adds a new method named `getTime` that gets the current time in the countdown. Then the `LaunchEvent` class checks the countdown time every 100 milliseconds and triggers the events only when the countdown clock actually says that it's time. This version of the application runs consistently.

In addition, you want to enable the `LaunchEvent` class to monitor the status of the `CountDownClock`, but you don't want to couple the `LaunchEvent` and `CountDownClock` classes too closely. Suppose that later, you develop a better countdown clock. If the `LaunchEvent` class knows what class is doing the counting, you have to recompile it if you use a different countdown class.

The solution is to use an interface as a buffer between the classes. This interface defines a method that gets the current status of the clock. Then the `CountDownClock` class can implement this interface, and the `LaunchEvent` class can use any object that implements this interface to get the time.

LISTING 1-4: **The Coordinated Countdown Application**

```
import java.util.ArrayList;                                          →1
public class CountDownApp
{
    public static void main(String[] args)
    {
        CountDownClock clock = new CountDownClock(20);               →7

        ArrayList<Runnable> events = new ArrayList<Runnable>();      →9
        events.add(new LaunchEvent(16, "Flood the pad!", clock));    →10
        events.add(new LaunchEvent(6, "Start engines!", clock));
        events.add(new LaunchEvent(0, "Liftoff!", clock));

        new Thread(clock).start();                                   →14

        for (Runnable e : events)                                    →16
        {
            new Thread(e).start();
        }
    }
}

interface TimeMonitor                                                →23
{
    int getTime();
}
```

```
class CountDownClock implements Runnable, TimeMonitor                    →28
{
    private int t;                                                        →30

    public CountDownClock(int start)                                      →32
    {
        this.t = start;
    }

    public void run()
    {
        for (; t >= 0; t--)                                               →39
        {
            System.out.println("T minus " + t);
            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {}
        }
    }

    public int getTime()                                                  →51
    {
        return t;
    }
}

class LaunchEvent implements Runnable                                     →57
{
    private int start;
    private String message;
    TimeMonitor tm;                                                       →61

    public LaunchEvent(int start, String message, TimeMonitor monitor)
    {
        this.start = start;
        this.message = message;
        this.tm = monitor;
    }

    public void run()
    {
        boolean eventDone = false;
        while (!eventDone)                                                →73
        {
            try
```

(continued)

LISTING 1-4: *(continued)*

```
            {
                Thread.sleep(100);
            }
            catch (InterruptedException e)
            {}
            if (tm.getTime() <= start)                                    →81
            {
                System.out.println(this.message);
                eventDone = true;
            }
        }
    }
}
```

The following paragraphs describe the high points of this version:

- **→1:** This import is required because the program uses the `ArrayList` class.
- **→7:** As you see in line 32, the constructor for the `CountDownClock` class now accepts a parameter to specify the starting time for the countdown. As a result, this line specifies `20` as the starting time for the `CountDownClock` object.
- **→9:** An `ArrayList` of `Runnable` objects is used to store each launch event.
- **→10:** The lines that create the `LaunchEvent` objects pass the `CountDownClock` object as a parameter to the `LaunchEvent` constructor. That way the `LaunchEvent` objects can call the clock's `getTime` method if necessary.
- **→14:** The clock is started! Note that this `CountDownClock` class is started by creating a new thread. That's because in this version, the `CountDownClock` class implements the `Runnable` interface instead of extending the `Thread` class.
- **→16:** An enhanced `for` loop starts threads to run the `LaunchEvent` objects.
- **→23:** The `TimeMonitor` interface defines just one method, named `getTime`. This method returns an integer that represents the number of seconds left on the countdown timer.
- **→28:** The `CountDownClock` class implements both the `Runnable` and the `TimeMonitor` interfaces.
- **→30:** A private field named `t` is used to store the current value of the countdown clock. That way, the current clock value can be accessed by the constructor, the `run` method, and the `getTime` method.

- **→32:** The constructor for the `CountDownClock` class accepts the starting time for the countdown as a parameter. Thus, this countdown clock doesn't have to start at 20 seconds. The value passed via this parameter is saved in the `t` field.
- **→39:** The `for` loop in the `run` method tests and decrements the `t` variable. But because this variable is already initialized, it doesn't have an initialization expression.
- **→51:** The `getTime()` method simply returns the value of the `t` variable.
- **→57:** This line is the start of the `LaunchEvent` class.
- **→61:** A private field of type `TimeMonitor` is used to access the countdown clock. A reference to this object is passed to the `LaunchEvent` class via its constructor. The constructor simply stores that reference in this field.
- **→73:** The `while` loop includes a call to `Thread.sleep` that sleeps for just 100 milliseconds. Thus, this loop checks the countdown clock every 100 milliseconds to see whether its time has arrived.
- **→81:** This statement calls the `getTime` method of the countdown clock to see whether it's time to start the event. If so, a message is displayed, and `eventDone` is set to true to terminate the thread.

This version of the countdown application is almost perfect, but it contains a subtle flaw that can still cause it to run inconsistently. In other words, this version of the program might still sometimes announce "Flood the pad!" *before* "T-minus 16" instead of after. I discuss this flaw and show you several ways to fix it in the next section.

Considering Thread Safety

Whenever you create a program that uses more than one thread, you must take pains to ensure that your threads play nice together. The issue is that a Java method may run multiple times simultaneously in different threads. If you aren't careful, those concurrent executions can interfere with each other in ways that break your code.

If you intend for a Java method to be executed in multiple threads, you must ensure that the method is *thread safe*, which simply means that you can count on the method behaving consistently and correctly no matter how many simultaneous threads call the method. Ensuring thread safety is the most difficult and important task of working with threads.

Here's a simple example of a method that is *not* thread safe:

```
int counter = 0;

void incrementCounter()
{
    counter++;
}
```

In this example, the `incrementCounter` method is not thread safe because the increment operator, when applied to an `int` variable, is itself not thread safe. To understand why, you need to realize that incrementing an `int` variable actually requires three operations: First, the current value must be retrieved from memory. Second, the value 1 must be added to the retrieved value. And third, the updated value must be written back to memory.

Suppose two threads call `incrementCounter()` at the same time. Both would retrieve the value of `counter` from memory — let's say it's 100. Both will add 1 to 100, producing 101. And then both will write 101 back to memory. Even though `counter` was incremented twice (once by each thread), the final result will be 101, not 102. In other words, one of the increment operations is lost because the individual steps of the increment operation can be interleaved in unpredictable ways when run by multiple threads.

Fortunately, there are ways to make methods like `IncrementCounter` thread safe, including atomic variables, the `synchronized` keyword, and the `ReentrantLock` class. You'll learn about each of these techniques later in this chapter. For now, just hang on to the core idea: Whenever you work with multiple threads, you must keep thread safety in mind.

Now let's turn our attention to the CountDownApp that was shown in Listing 1-4. Is it thread safe? The answer is a resounding no. This app is not thread safe because multiple threads running `LaunchEvent` instances depend on access to accurate countdown times from the thread that runs the `CountDownClock`, and the `CountDownClock` class does not take measures to ensure that those threads always receive the most up-to-date countdown time.

Here's the root of the problem: The field `t`, defined in line 30, maintains the countdown time in the `CountDownClock` class. Although only a single instance of this class is created, a reference to that instance (`clock`) is passed to each of the three `LaunchEvent` threads. All three of the `LaunchEvent` threads have access to the clock's `getTime` method, which in turn accesses the field `t`. Thus, all three of the `LaunchEvent` threads indirectly read the value of `t`.

It's entirely possible that one of the `LaunchEvent` threads could read an older value rather than the current value of `t`. If that were to happen, the launch event might fire just an instant later than expected. (Given how slow the countdown is in computer terms, this is extremely unlikely — but still possible.)

The solution is to ensure that all of the launch event threads always receive the most currently updated value of the shared field `t`. Fortunately, Java provides several ways to do this:

- The `volatile` keyword
- The `AtomicInteger` class
- The `synchronized` keyword
- The `ReentrantLock` class

The next four sections describe each of these techniques in detail. And each section presents an alternate version of the `CountDownClock` class that uses one of these techniques to ensure thread safety.

Using the volatile Keyword for Thread Safety

Normally, every thread that accesses a shared field keeps its own cached copy of the field and periodically refreshes the field's value. If you add the `volatile` keyword to the shared field's declaration, other threads are not allowed to cache the field's value. That way, they will always work with the most current value of the shared field.

You can completely solve the thread safety problem of the countdown app shown in Listing 1-4 by adding `volatile` to the declaration of the `t` field, as shown in Listing 1-5. With this one simple change, the countdown app is thread safe. And the performance cost is small. Although threads that access the volatile shared value miss out on the performance benefits of caching the data, but that's a small trade in exchange for always using the correct value.

LISTING 1-5: **A Thread-Safe Version of the CountDownClock Class That Uses the volatile Keyword**

```
class CountDownClock implements Runnable, TimeMonitor
{
    private volatile int t;                                                    →3

    public CountDownClock(int start)
    {
        this.t = start;
    }

    public void run()
    {
        for (; t >= 0; t--)
        {
            System.out.println("T minus " + t);
            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {}
        }
    }

    public int getTime()
    {
        return t;
    }
}
```

The only line in this version of the class that differs from the version shown in Listing 1-4 is:

» →3 Here, the shared field `t` is declared using the `volatile` keyword, which prevents other threads from caching its value. This single change is all that's needed to make the class thread safe.

Here are some additional details you should know about using `volatile`:

» `volatile` is intended for applications in which a shared field is updated by only one thread but might be accessed by multiple threads. If the field might be updated by more than one thread, `volatile` won't make the field thread safe. Instead, you should use an atomic data type or synchronize the method.

- `volatile` should be used only for shared variables declared at the class scope — in other words, for fields. You don't need to use `volatile` for variables declared within a method.
- If you have two or more shared fields that must be updated together, marking them both as `volatile` won't work. For example, suppose you have `int` fields named `low` and `high`, and they must always be updated as a set. Marking them both as `volatile` will ensure that other threads don't cache the values. But suppose Thread A reads the value of `low`, but before it can read the value of `high`, Thread B updates both values. Now Thread A has the updated value of `high` but an outdated value of `low`. To prevent this from happening, you'll need to use the `synchronized` keyword, as described in the section "Using the `synchronized` Keyword for Thread Safety."
- The specific problem that the `volatile` keyword solves is called *visibility*. Visibility ensures that all threads see the most current value of a shared field, and that no thread holds on to an outdated value of a shared field.



TECHNICAL STUFF



TIP

For more information about the word *volatile*, see the sidebar, "Volatile Is Such a Flighty Word."

VOLATILE IS SUCH A FLIGHTY WORD

In Java, the word *volatile* has a very specific meaning: It designates a field whose value may change at any time and must always be read fresh from memory whenever any thread needs to access the value. Another way to put it is that the value of any field marked with the volatile keyword must never be cached.

In modern parlance, the word *volatile* almost always has a negative connotation. Uncertain stock markets are volatile. People who are prone to violent outbursts are volatile. Chemicals that may explode are volatile. None of these are seen as desirable traits.

But in Java, the word volatile is not judgmental. Whether a shared field might change is neither good nor bad—it just *is*. You're not insulting a field when you call it volatile; you're just stating a fact about the field that others need to be aware of.

Interestingly, the word *volatile* has not always implied instability, at least not in a pejorative sense. The origin of the word is actually fascinating. One of its earliest uses is found

(continued)

(continued)

in a poem from the late 13th century, when the word was a synonym for *birds*. The poem describes a variety of game birds as follows:

> Partyhches, ployers, and heroun . . .
>
> larkes and smale volatyle.

Translated to modern spelling, it reads "Partridges, plovers, and heron . . . larks and small volatile," That is, "small birds."

It didn't take long for the word *volatile* to come to mean bird-like behavior: flying about, acting unpredictably, difficult to pin down, and so on. In Shakespeare's final play *The Two Noble Kinsmen* (co-written with John Fletcher around 1613), a physician describes the mental instability of another character as "a volatyle skipping spirit." This is the earliest published use of the word with a negative connotation.

The word entered the field of science in the 17th century when chemists coopted it to describe substances that easily evaporate or can't be confined. From there, the word took on its general meaning where it can be applied to virtually anything that is unstable, unpredictable, or prone to erratic changes.

Use the `volatile` keyword in Java without judgment. But when you do, think of the word's 700-year history of describing things that refuse to stay put: birds, emotions, unstable compounds, stock markets, and shared fields in threaded Java programs.

Using the AtomicInteger Class for Thread Safety

The volatile keyword isn't the only way you can provide Thread Safety for the countdown application. Another method is to use a class called `AtomicInteger` as the type for the shared field `t`. When you use the `AtomicInteger` class, the field is automatically treated as `volatile`. In addition, `AtomicInteger` guarantees that a *compound operation* — one that reads a value, changes it, and writes it back, such as decrementing `t` by `1` — happens as a single, uninterruptible step.

That's important because a compound operation is exactly where the classic non-thread-safe `counter++` example goes wrong: Two threads read the same starting value, each thread computes its own update, and each thread writes back. Without thread safety, one of those two updates is silently lost. If `counter` starts at `10`,

and two threads increment it at the same time, there's a real risk that the ending `counter` value will end up being 11, not 12.

That's where `AtomicInteger` comes in. It ensures that when the increment operation begins, no other thread can access the field's value until the operation completes. In the context of thread safety, an *atomic operation* is an operation that is indivisible and uninterruptable. An atomic operation may consist of several steps, but those steps can't be divided. From the perspective of other threads, an atomic operation behaves as if it were a single step.

The bottom line is that when you increment an `AtomicInteger`, it is not possible for another thread to access the value until after the increment operation completes. As a result, `AtomicInteger` ensures thread safety for an operation like increment: If `counter` is an `AtomicInteger` rather than an `int`, its value is 10, and if two threads increment it, the result will *always* be 12.

In terms of performance, the extra cost of using `AtomicInteger` rather than `int` is negligible. However, writing code that uses `AtomicInteger` is more complex because `AtomicInteger` is not a primitive data type. For starters, you can't just declare a variable as an `AtomicInteger` and assign it a value in the same way you would with an `int` variable. Instead, you must use the `new` keyword and pass the initial value as an argument in the constructor:

```
AtomicInteger counter = new AtomicInteger(0);
```

After you create an `AtomicInteger`, you can't use ordinary expression operators like `++`, `+`, `-`, `*`, or `/`. Instead, you must use the various methods of the `AtomicInteger` class to perform basic math. For example, here's a method that increments counter:

```
void incrementCounter()
{
    counter.incrementAndGet();
}
```

Here, the `incrementAndGet()` method increments the counter value and returns the result.

The `AtomicInteger` class is one of several atomic data types provided by a package called `java.util.concurrent.atomic`, which you must import to use it in your programs. This package provides several other atomic data types, including `AtomicBoolean` and `AtomicLong`, but I focus in this chapter on `AtomicInteger` because it's the most commonly used.

Table 1-2 list the most commonly used constructors and methods of the AtomicInteger class. Notice that the methods are limited to basic increment, decrement, and addition operations. Also, note that each method is available in two forms: one that returns the updated value (after the operation is applied), and another that returns the original value (before the operation is applied). You'll probably use the version that returns the updated value more often than the version that returns the original value.

TABLE 1-2 **Constructors and Methods of the AtomicInteger Class**

Constructor	Explanation
`AtomicInteger()`	Creates a new `AtomicInteger` with an initial value of 0.
`AtomicInteger (int initialValue)`	Creates a new `AtomicInteger` with the specified initial value.
Method	**Explanation**
`int addAndGet(int delta)`	Atomically adds the specified value and returns the new value.
`int decrementAndGet()`	Atomically subtracts one and returns the new value.
`double doubleValue()`	Returns the value as a `double`.
`float floatValue()`	Returns the value as a `float`.
`int get()`	Returns the value as an `int`.
`int getAndDecrement()`	Atomically subtracts 1 from the value, but returns the original value.
`int getAndIncrement()`	Atomically adds 1 to the value, but returns the original value.
`int getAndSet(int newValue)`	Assigns a new value and returns the original value.
`int incrementAndGet()`	Atomically adds 1 to the value and returns the new value.
`int intValue()`	Returns the value as an `int`. This method is identical to `get()`.
`long longValue()`	Returns the value as a `long`.
`void set(int newValue)`	Assigns a new value.

To use the AtomicInteger class to make the CountDownClock class thread-safe, you simply convert the `t` field from an `int` to an AtomicInteger. Listing 1-6 shows the CountDownClock class with these changes. Note that the CountDownApp class and the LaunchEvent class are not changed from Listing 1-4 and so are not repeated in Listing 1-6. In other words, making the CountDownClass thread safe makes the entire application thread safe.

LISTING 1-6: **A Thread--Safe Version of the CountDownClock Class That Uses the AtomicInteger Class**

```
import java.util.concurrent.atomic.AtomicInteger;                    →1

class CountDownClock implements Runnable, TimeMonitor
{
    private AtomicInteger t;                                         →5

    public CountDownClock(int start)
    {
        this.t = new AtomicInteger(start);                           →9
    }

    public void run()
    {
        for (; t.get() >= 0; t.decrementAndGet())                    →14
        {
            System.out.println("T minus " + t.get());                →16
            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {}
        }
    }
    public int getTime()
    {
        return t.get();                                              →27
    }
}
```

Here are the changes required to implement the `CountDownClock` class with an `AtomicInteger` rather than an `int`:

- **→1:** This `import` statement is required to use the `AtomicInteger` class.
- **→5:** The shared field `t` is declared as an `AtomicInteger`.
- **→9:** In the `CountDownClass` constructor, an instance of the `AtomicInteger` class is created using the `start` argument. This new instance is assigned to the shared field `t`.

- **→14:** The `for` loop is rewritten here to use the methods of the `AtomicInteger` class.
- **→16:** `System.out.println` uses `t.get()` rather than just `t` to print the current countdown value.
- **→27:** Similarly, the `getTime()` method uses `t.get()` rather than just `t` to return the current value of the countdown timer.

Using the synchronized Keyword for Thread Safety

So far in this chapter, you've seen two ways to provide thread safety for Java classes: The `volatile` keyword and the `AtomicInteger` class. Both of those techniques apply to individual shared fields that are vulnerable to thread safety issues. In this section, I show you how to use the `synchronized` keyword, which raises the level of focus up from an individual field to a specific block of code or to an entire method. When you use the `synchronized` keyword, you're telling Java that only one thread at a time can run the code that's within the synchronized scope. Any other thread that attempts to run the synchronized code will have to wait until the first thread finishes.

From a performance perspective, `synchronized` is relatively expensive. Under the hood, there's a lot going on: creating locks, monitoring them, releasing them at the right time, and more. That's a lot of overhead. However, at the coding level, it's as simple as placing the `synchronized` keyword at the right spots in your code. And when you rely on `synchronized` to protect the code that accesses or manipulates your shared fields, you don't need to use `volatile` or `AtomicInteger` to protect those fields.

To synchronize an entire method, just add the keyword `synchronized` to the method header. For example, here's a synchronized version of the `getTime()` method:

```
public synchronized int getTime()
{
    return t;
}
```

If you don't need to synchronize an entire method, you can synchronize just a block of code. The only trick when you do that is that you must provide an object

reference that Java can apply the synchronization to. Most of the time, you can just specify `this`. Here's an example that decrements the countdown clock:

```
synchronize(this)
{
    t--;
}
```

Here, the decrement operation — which is actually a three-step operation (get the value from memory, subtract 1 from the value, and write the value back to memory) — is synchronized so that no other thread can decrement `t` until this thread has finished its decrement operation.

That's about all there is to it. Listing 1-7 shows a version of the `CountDownClass` that uses the `synchronized` keyword to ensure thread safety.

LISTING 1-7: **Using synchronized in the CountDown Class**

```
class CountDownClock implements Runnable, TimeMonitor
{
    private int t;

    public CountDownClock(int start)
    {
        this.t = start;
    }

    public void run()
    {
        boolean stillCounting = true;

        while (stillCounting)
        {
            synchronized(this)                                          →16
            {
                System.out.println("T minus " + t);
                stillCounting = (t > 0);
                t--;
            }
            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {}
        }
    }
```

(continued)

LISTING 1-7: *(continued)*

```
    public synchronized int getTime()                                    →31
    {
        return t;
    }
}
```

Here are the changes required to implement the `CountDownClock` class with `synchronized`:

- **→16:** This code block is synchronized so that the print, the `stillCounting` check, and the decrement of `t` all happen as a single, uninterruptible unit. This ensures that no other thread can read or change the value of `t` while these three statements are executing.
- **→31:** The `getTime` method is synchronized to ensure that it always returns the most up-to-date value of `t`.



It's worth noting that the countdown application is not really a good candidate for using `synchronized`. Synchronizing is a bit of a drastic step that imposes a significant performance penalty. The extra overhead of `synchronized` makes sense when a lot of related fields need to be kept consistent, but not when just one field (in this case, `t`) is involved.

Using the ReentrantLock Class for Thread Safety

So far, we've considered three ways to create thread-safe methods: using the `volatile` keyword, the `AtomicInteger` class, and the `synchronized` keyword. In this section, I explain a fourth method: explicitly creating, applying, and releasing your own locks to ensure that the specific areas of your code are thread safe. To do that, you use the `ReentrantLock` class.

The `ReentrantLock` class has quite a few methods, but I delve into just two of them in this chapter:

- `lock()`, which acquires a lock.
- `unlock()`, which releases the lock.

Here's an example of how you can create, acquire, and release a lock:

```
ReentrantLock lock = new ReentrantLock();
lock.lock();
// Place the code you want locked here
lock.unlock();
```

After a thread acquires a lock by calling the lock's `lock()` method, the lock won't let any other thread acquire that lock until the thread that currently has the lock calls the `unlock()` method to release the lock. If Thread A has acquired a lock and then Thread B tries to acquire the same lock, Thread B is suspended until Thread A releases the lock. After Thread A releases the lock, Thread B gets the lock and can proceed.

The most interesting — and important — thing about a `ReentrantLock` is that it doesn't know or care about what you're actually trying to lock. The other three thread-safe techniques I've presented so far in this chapter all apply to specific entities: `volatile` applies to a specific variable, `AtomicInteger` applies to a specific object, and `synchronized` applies to a specific block of code. But `ReentrantLock` doesn't apply to anything other than the lock itself.

That's actually the key point about `ReentrantLock`: The only thing it locks is the lock itself. After you've acquired the lock, your code can do anything you want it to do, with the guarantee that everything your code does until it releases the lock is guaranteed to be thread safe.

`ReentrantLock` is very powerful, and very dangerous. If you were to ask me, "What is the greatest locking commandment?" I would answer: "The first is that whenever you acquire a lock, you must release it as quickly as you can. And the second is just like it: If you acquire a lock, you must never, ever, under any circumstances, forget to release it."

You may be wondering, then, how can you use `ReentrantLock` in a method like the `CountDownClock.getTime()` method, which simply returns the current value of the internal clock. You may be tempted to write that method like this:

```
public  int getTime()
{
    lock.lock();
    return t;
    lock.unlock();
}
```

But that code won't compile: The `lock.unlock()` line will be flagged as unreachable code. If you simply omit the line, the lock will never be released.

So, what's the fix? The `finally` block of the familiar `try-catch-finally` pattern is the answer. Write the method like this:

```
public  int getTime()
{
    lock.lock();
    try
    {
        return t;
    }
    finally
    {
        lock.unlock();
    }
}
```

This works because the code in the `finally` block is guaranteed to run — even if the `return` statement in the `try` block is successful.

Listing 1-8 shows a complete version of the `CountDownClock` class that uses `ReentrantLock` to ensure thread safety.

LISTING 1-8:

A Thread-Safe Version of the CountDownClock Class That Uses the ReentrantLock Class

```
import java.util.concurrent.locks.ReentrantLock;                    →1

class CountDownClock implements Runnable, TimeMonitor
{
    private int t;                                                  →5
    ReentrantLock lock = new ReentrantLock();                       →6

    public CountDownClock(int start)
    {
        this.t = start;
    }

    public void run()
    {
        boolean stillCounting = true;
        while (stillCounting)
        {
            lock.lock();                                            →18
            System.out.println("T minus " + t);
            stillCounting = (t > 0);
```

```
            t--;
            lock.unlock();                                          →22

            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {}
        }
    }

    public  int getTime()
    {
        lock.lock();                                                →35
        try
        {
            return t;
        }
        finally
        {
            lock.unlock();                                          →42
        }
    }
}
```

Here are the salient points of this listing:

- **→1:** This `import` statement is required to use `ReentrantLock`.
- **→5:** The shared field `t` is defined as an `int`. The `volatile` keyword is not required because this version uses a `ReentrantLock` to ensure that all code that accesses or updates this variable is locked.
- **→6:** This line creates a new `ReentrantLock` object and assigns it to a class variable named `lock`.
- **→18:** The lock is acquired prior to manipulating the countdown timer.
- **→22:** The lock is released after the time is decremented.
- **→35:** The lock is acquired again prior to returning the current countdown time in the `getTime()` method.
- **→42:** The lock is released in the `finally` block, ensuring that the lock is promptly released when the countdown value is returned.

Coping with Threadus Interruptus

You can interrupt another thread by calling its `interrupt` method, provided that you have a reference to the thread, as in this example:

```
t.interrupt();
```

Here the thread referenced by the `t` variable is interrupted. Now all the interrupted thread has to do is find out that it has been interrupted and respond accordingly. That's the topic of the following sections.

Finding out whether you've been interrupted

As you've already seen, several methods of the `Thread` class, including `sleep` and `yield`, throw `InterruptedException`. Up until now, I've told you to simply ignore this exception — and in many cases, that's appropriate. Many (if not most) threads, however, should respond to `InterruptedException` in one way or another. In most cases, the thread should terminate when it's interrupted.

Unfortunately, finding out whether a thread has been interrupted isn't as easy as it sounds. `InterruptedException` is thrown when another thread calls the `interrupt` method on this thread while the thread is not executing. That's why the methods that can cause the thread to give up control to another thread throw this exception. That way, when the thread resumes execution, you know that it was interrupted.

The `yield` and `sleep` methods aren't the only way for control to be wrested away from a thread, however. Sometimes the thread scheduler just steps in and says, "You've had enough time; now it's someone else's turn to play." If that happens and then some other thread calls your thread's `interrupt` method, `InterruptedException` isn't thrown. Instead, a special flag called the *interrupted flag* is set to indicate that the thread was interrupted.

You can test the status of the interrupted flag in one of two ways: You can call the `isInterrupted()` method, which returns `true` if the thread was interrupted. Or, you can call the `interrupted()` method, which not only returns the status of the interrupted flag, but also resets it to `false`. Use whichever method suits your situation.

Either way, your threads have to check twice to see whether they have been interrupted: first by catching `InterruptedException`, and second by checking the interrupted flag. The usual way to do that is to follow this form:

```
public void run()
{
    boolean done    = false;     // set done=true when the thread completes
                                 // its work without being interrupted
    boolean aborted = false;     // set abort=true if the thread is
                                 // interrupted
    // Note that the thread ends when either done or abort becomes true.
    while(!done)
    {
        // do the thread's work here
        // set done=true when the thread's work is finished.
        try
        {
            Thread.sleep(100);   // sleep a bit
        }
        catch(InterruptedException e)
        {
            aborted = true;
        }
        if (Thread.interrupted())
            aborted = true;
        if (aborted)
            break;
    }
}
```

Here the `boolean` variable `aborted` is set to `true` if `InterruptedException` is thrown or if the `interrupted` flag is set. Then, if `aborted` has been set to true, a `break` statement is executed to leave the `while` loop, thus ending the thread.

Aborting the countdown

To illustrate how you can interrupt threads, Listing 1-9 shows yet another version of the countdown application. This version aborts the countdown if something goes wrong with any of the launch events. (As a heads-up, this program is based on the version that was shown in Listing 1-4 but with the `volatile` keyword added to ensure thread safety.)

To simplify the code a bit, I assume that things aren't going well at NASA, so every launch event results in a failure that indicates a need to abort the countdown. Thus, whenever the start time for a `LaunchEvent` arrives, the `LaunchEvent`

class attempts to abort the countdown. Thus, the first `LaunchEvent` that reaches its start time aborts the launch — in our case, it's the Flood the Pads! Launch event that triggers the abort. (It goes without saying that in a real launch-control program, you wouldn't want to abort the launch unless something actually *did* go wrong.)

LISTING 1-9: **The Countdown Application with Aborts**

```
import java.util.ArrayList;

public class CountDownAbortApp                                          →3
{
    public static void main(String[] args)
    {
        CountDownClock clock = new CountDownClock(20);
        ArrayList<Runnable> events =
            new ArrayList<Runnable>();
        events.add(new LaunchEvent(16, "Flood the pad!", clock));
        events.add(new LaunchEvent(6, "Start engines!", clock));
        events.add(new LaunchEvent(0, "Liftoff!", clock));
        new Thread(clock).start();
        for (Runnable e : events)
            new Thread(e).start();
    }
}

interface TimeMonitor
{
    int getTime();
    void abortCountDown();                                              →22
}

class CountDownClock implements Runnable, TimeMonitor
{
    private volatile int t;

    public CountDownClock(int start)
    {
        this.t = start;
    }

    public void run()
    {
        boolean aborted = false;                                        →36
        for (; t >= 0; t--)
        {
            System.out.println("T minus " + t);
```

```
            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {
                aborted = true;                                          →46
            }
            if (Thread.interrupted())
            {
                aborted = true;                                          →50
            }
            if (aborted)                                                 →52
            {
                System.out.println(
                    "Stopping the clock!");
                break;
            }
        }
    }

    public int getTime()
    {
        return t;
    }

    public synchronized void abortCountDown()                            →66
    {
        Thread[] threads = new Thread[Thread.activeCount()];             →68
        Thread.enumerate(threads);                                       →69
        for(Thread t : threads)                                          →70
        {
            t.interrupt();
        }
    }
}

class LaunchEvent implements Runnable
{
    private int start;
    private String message;
    TimeMonitor tm;

    public LaunchEvent(int start, String message, TimeMonitor monitor)
    {
        this.start = start;
        this.message = message;
        this.tm = monitor;
```

(continued)

LISTING 1-9: *(continued)*

```
    }

    public void run()
    {
        boolean eventDone = false;
        boolean aborted = false;                                          →93
        while (!eventDone)
        {
            try
            {
                Thread.sleep(10);
            }
            catch (InterruptedException e)
            {
                aborted = true;                                           →102
            }
            if (tm.getTime() <= start)
            {
                System.out.println(this.message);
                eventDone = true;
                System.out.println("ABORT!!!!");
                tm.abortCountDown();
            }
            if (Thread.interrupted())
            {
                aborted = true;                                           →113
            }
            if (aborted)                                                  →115
            {
                System.out.println(
                    "Aborting " + message);
                break;
            }
        }
    }
}
```

The following paragraphs point out the highlights of this program:

» **→3:** The `CountDownApp` class itself hasn't changed. That's the beauty of object-oriented programming. Although I changed the implementations of the `CountDownClock` and `LaunchEvent` classes, I didn't change the public interfaces for these classes. As a result, no changes are needed in the `CountDownApp` class.

- **→22:** The `LaunchEvent` class needs a way to notify the `CountDownClock` class that the countdown should be aborted. To do that, I added an `abortCountDown` method to the `TimeMonitor` interface.
- **→36:** The `run` method of the `CountDownClock` class uses a `boolean` variable named `aborted` to indicate whether the thread has been interrupted. This variable is set to `true` in line 46 if `InterruptedException` is caught. It's also set to `true` in line 50 if `Thread.interrupted()` returns `true`.
- **→52:** If the `aborted` field has been set to `true`, it means that the thread has been interrupted, so the message `Stopping the clock!` is displayed, and a `break` statement exits the loop. Thus the thread is terminated.
- **→66:** The `abortCountDown` method is synchronized. That's required because any of the `LaunchEvent` objects can call it, and there's no guarantee that they won't all try to call it at the same time.
- **→68:** The `abortCountDown` method starts by creating an array of `Thread` objects that's large enough to hold all the active threads. The number of active threads is provided by the `activeCount` method of the `Thread` class.
- **→69:** The `abortCountDown` method calls the `enumerate` method of the `Thread` class to copy all the active threads into this array. Note that this method is static, so you don't need a reference to any particular thread to use it. (The `activeCount` method used in line 68 is static too.)
- **→70:** An enhanced `for` loop is used to call the `interrupt` method on all the active threads. That method shuts down everything.
- **→93:** Like the `CountDownClock` class, the `LaunchEvent` class uses a `boolean` variable to indicate whether the thread has been interrupted. This variable is set if `InterruptedException` is caught in line 102 or if `Thread.interrupted()` returns `true` in line 113; then it's tested in line 115. If the `aborted` variable has been set to `true`, the thread prints a message indicating that the launch event has been aborted, and a `break` statement is used to exit the loop and (therefore) terminate the thread.

When you run this version of the countdown application, the console output will appear something like this (minor variations might occur because of the synchronization of the threads):

```
T minus 20
T minus 19
T minus 18
T minus 17
T minus 16
Flood the pad!
ABORT!!!!
```

```
Stopping the clock!
Aborting Flood the pad!
Aborting Start engines!
Aborting Liftoff!
```

Considering Virtual Threads

The Java thread mechanisms I've covered so far in this chapter are incredibly useful, but they're also costly from a performance perspective. That's because these threads rely on the underlying operating system's threading features. Thus, whenever you create a thread, the JVM has to coordinate with the operating system to create and manage the thread.

Beginning in 2017, Java's developers began considering a more efficient way to create and manage threads, in which the JVM would handle the scheduling and execution of the threads, not the operating system. The intent was to make large-scale threading more practical. And by "large-scale threading," I mean applications that use thousands or even millions of threads.

The new feature, called Virtual Threads, was first introduced in September 2022 with Java 19 and, in September 2023, with Java 21, it became an official feature. Along with the introduction of Virtual Threads came a new term, *Platform Threads*, to describe the traditional thread features I've covered so far in this chapter.

The good news is that programming virtual threads is pretty much the same as programming platform threads. In fact, you can change the countdown app so that the launch event threads are virtual threads by changing only one line of code in the `main` method. Instead of:

```
new Thread(e).start();
```

Use this:

```
Thread.startVirtualThread(e);
```

This starts the `LaunchEvent` threads as Virtual Threads rather than Platform Threads.

Listing 1-10 shows the main method for the countdown application, modified so that the `LaunchEvent` threads are virtual threads. The only change between this version and the version presented in previous listings in this chapter is how the threads are created in line 14.

LISTING 1-10: The Main Method of the Countdown Application with Virtual Threads

```
import java.util.ArrayList;

public class CountDownVirtualApp
{
    public static void main(String[] args)
    {
        CountDownClock clock = new CountDownClock(20);
        ArrayList<Runnable> events = new ArrayList<Runnable>();
        events.add(new LaunchEvent(16, "Flood the pad!", clock));
        events.add(new LaunchEvent(6, "Start engines!", clock));
        events.add(new LaunchEvent(0, "Liftoff!", clock));
        new Thread(clock).start();
        for (Runnable e : events)
            Thread.startVirtualThread(e);                                    →14
    }
}
```