23.1 Introduction

[*Note: Sections marked "Advanced" are intended for readers who wish a deeper treatment of concurrency and may be skipped by readers preferring only basic coverage.*] It would be nice if we could focus our attention on performing only one task at a time and doing it well. That's usually difficult to do in a complex world in which there's so much going on at once. This chapter presents Java's capabilities for creating and managing multiple tasks. As we'll demonstrate, this can greatly improve program performance and responsiveness.

When we say that two tasks are operating **concurrently**, we mean that they're both *making progress* at once. Until the early 2000s, most computers had only a single processor. Operating systems on such computers execute tasks concurrently by rapidly switching between them, doing a small portion of each before moving on to the next, so that all tasks keep progressing. For example, it's common for personal computers to compile a program, send a file to a printer, receive electronic mail messages over a network and more, concurrently. Since its inception, Java has supported concurrency.

When we say that two tasks are operating **in parallel**, we mean that they're executing *simultaneously*. In this sense, parallelism is a subset of concurrency. The human body performs a great variety of operations in parallel. Respiration,

blood circulation, digestion, thinking and walking, for example, can occur in parallel, as can all the senses—sight, hearing, touch, smell and taste. It's believed that this parallelism is possible because the human brain is thought to contain billions of "processors." Today's multi-core computers have multiple processors that can perform tasks in parallel.

Java Concurrency

Java makes concurrency available to you through the language and APIs. Java programs can have multiple **threads of execution**, each with its own method-call stack and program counter, allowing it to execute concurrently with other threads while sharing with them application-wide resources such as memory and file handles. This capability is called **multithreading**.

Performance Tip 23.1

A problem with single-threaded applications that can lead to poor responsiveness is that lengthy activities must complete before *others can begin. In a multithreaded application, threads can be distributed across multiple cores (if available) so that multiple tasks execute in parallel and the application can operate more efficiently. Multithreading can also increase performance on single-processor systems—when one thread cannot proceed (because, for example, it's waiting for the*

result of an I/O operation), another can use the processor.

Concurrent Programming Uses

We'll discuss many applications of **concurrent programming**. For example, when streaming an audio or video over the Internet, the user may not want to wait until the entire audio or video downloads before starting the playback. To solve this problem, multiple threads can be used—one to download the audio or video (later in the chapter we'll refer to this as a *producer*), and another to play it (later in the chapter we'll refer to this as a *consumer*). These activities proceed concurrently. To avoid choppy playback, the threads are **synchronized** (that is, their actions are coordinated) so that the player thread doesn't begin until there's a sufficient amount of the audio or video in memory to keep the player thread busy. Producer and consumer threads *share memory*—we'll show how to coordinate these threads to ensure correct execution. The Java Virtual Machine (JVM) creates threads to run programs and threads to perform housekeeping tasks such as garbage collection.

Concurrent Programming Is Difficult

Writing multithreaded programs can be tricky. Although the

human mind can perform functions concurrently, people find it difficult to jump between parallel trains of thought. To see why multithreaded programs can be difficult to write and understand, try the following experiment: Open three books to page 1, and try reading the books concurrently. Read a few words from the first book, then a few from the second, then a few from the third, then loop back and read the next few words from the first book, and so on. After this experiment, you'll appreciate many of the challenges of multithreading—switching between the books, reading briefly, remembering your place in each book, moving the book you're reading closer so that you can see it and pushing the books you're not reading aside—and, amid all this chaos, trying to comprehend the content of the books!

Use the Prebuilt Classes of the Concurrency APIs Whenever Possible

Programming concurrent applications is difficult and error prone. If you must use synchronization in a program, follow these guidelines:

1. *The vast majority of programmers should use existing collection classes and interfaces from the concurrency APIs that manage synchronization for you—such as the* `ArrayBlockingQueue` *class (an implementation of interface* `BlockingQueue`*) we discuss in Section 23.6*. Two other concurrency API classes that you'll use frequently are `LinkedBlockingQueue` and `ConcurrentHashMap` (each summarized in Fig. 23.22). The concurrency API classes are written by

experts, have been thoroughly tested and debugged, operate efficiently and help you avoid common traps and pitfalls. Section 23.10 overviews Java's prebuilt concurrent collections.

2. For advanced programmers who want to control synchronization, use the `synchronized` keyword and `Object` methods `wait`, `notify` and `notifyAll`, which we discuss in the optional Section 23.7.
3. Only the most advanced programmers should use `Lock`s and `Condition`s, which we introduce in the optional Section 23.9, and classes like `LinkedTransfer-Queue`—an implementation of interface `TransferQueue`—which we summarize in Fig. 23.22.

You might want to read our discussions of the more advanced features mentioned in items 2 and 3, even though you most likely will not use them. We explain these because:

- They provide a solid basis for understanding how concurrent applications synchronize access to shared memory.
- By showing you the complexity involved in using these low-level features, we hope to impress upon you the message: *Use the simpler prebuilt concurrency capabilities whenever possible*.