Concurrency: Part I 22

Chapter Topics

- Distinguishing between concurrency and parallelism
- Understanding runtime organization for thread execution: what is shared memory and what is private to a thread
- Creating threads by extending the `Thread` class or by implementing the `Runnable` interface
- Writing synchronized code using `synchronized` methods and `synchronized` statements to achieve mutually exclusive access to shared resources
- Managing the thread lifecycle: thread states, the transitions between them, and thread coordination
- Understanding threading problems: liveness, fairness, deadlock, starvation, livelock, and memory consistency errors
- Understanding which thread behavior a program can take as guaranteed, and which behavior is not guaranteed, including the happens-before relationship

Java SE 17 Developer Exam Objectives

[8.1] Create worker threads using Runnable and Callable, manage the thread lifecycle, including automations provided by different Executor services and concurrent API	§22.1, p. 1367, *to* §22.5, p. 1408

- *Creating threads with the* `Runnable` *functional interface and managing the thread lifecycle are covered in this chapter.*
- *Creating worker threads with the* `Callable` *functional interface, and managing concurrency using executor services of the Concurrency API are covered in* §23.2, p. 1423.

Java SE 11 Developer Exam Objectives

[8.1] Create worker threads using Runnable and Callable, and manage concurrency using an ExecutorService and java.util.concurrent API	§22.3, p. 1371

- *Creating threads with the* `Runnable` *functional interface is covered in this chapter.*

Java SE 11 Developer Exam Objectives

- ❍ *Creating worker threads with the* `Callable` *functional interface, and managing concurrency using executor services of the Concurrency API are covered in* §23.2, p. 1423.

Support for multithreaded programming (also called *concurrent programming*) is an integral feature of the Java ecosystem to harness the computing power of multiprocessor and multi-core architectures. The programming language, the tools (particularly the JVM), and the APIs have all evolved to meet this challenge.

This chapter focuses on *low-level* support for multithreaded programming. This includes creating threads to execute tasks, understanding the thread lifecycle, how threads cooperate, and thread issues that are common in concurrent programming.

Chapter 23, p. 1419, deals with *high-level* support for building massively concurrent applications. This support is primarily provided by the frameworks in the `java.util.concurrent` package. High-level mechanisms allow the programmer to concentrate on *task management* in concurrent applications, while taking care of low-level *thread management*.

22.1 Threads and Concurrency

We first look at some basic concepts before diving into multithreaded programming in Java.

Multitasking

Multitasking allows several activities to occur concurrently on the computer. A distinction is usually made between:

- Process-based multitasking
- Thread-based multitasking (associated synonymously with *concurrency* in Java)

At the coarse-grain level there is *process-based* multitasking, which allows processes (i.e., programs) to run concurrently on the computer. A familiar example is running a spreadsheet program while also working with a word processor. At the fine-grain level there is *thread-based* multitasking, which allows parts of the *same* program to run concurrently on the computer. A familiar example is a word processor that is formatting text as it is typed and is spell-checking at the same time. This is only feasible if the two tasks are performed by two independent paths of execution at runtime. The two tasks would correspond to executing parts of the program code concurrently.

The sequence of code executed for each task defines an *independent sequential path of execution* within the program, and is called a *thread* (*of execution*). Many threads can run concurrently within a program, doing different tasks. At runtime, threads in a program exist in a

common memory space, and can therefore share both data and code (i.e., they are *light-weight* compared to processes). They also share the process running the program.

In a single-threaded environment only one task at a time can be performed. CPU cycles are wasted—for example, when waiting for user input. Multitasking allows idle CPU time to be put to good use.

Some advantages of thread-based multitasking as compared to process-based multitasking are:

- Threads share the same address space—that is, global data is accessible to all threads.
- Context switching between threads—that is, switching of execution from one thread to another—is usually less expensive than between processes.
- The cost of coordination between threads is relatively low.

Java supports thread-based multitasking—that is, concurrency—and provides support for multithreaded programming. *Thread-safety* is the term used to describe the design of classes that ensure that the state of their objects is always consistent, even when the objects are used concurrently by multiple threads.

Understanding Concurrency and Parallelism

There is an important distinction between the concept of *parallelism* and *concurrency* when it comes to code execution.

Parallel code execution implies *simultaneous* execution of instructions on different CPU cores. Concurrent execution is a more general idea that does not necessarily imply that the code of different threads is actually executed in parallel. It is possible to create more threads than the CPU cores can execute in parallel, which will cause threads to take turns at sharing time slices on the CPU cores. This means that sometimes the threads are running in parallel and sometimes it only feels like they are because they compete for time slices to execute on a limited number of CPU cores. In other words, concurrent execution gives an illusion of parallel execution.

The JVM is not the only process on the computer that requires CPU time. The operating system and other tasks also share CPU time, making it impossible to predict how much CPU time is allocated and when a thread will get to run, or if any given set of code will be executed in parallel at all. These factors have very important consequences:

1. Execution of code in different threads is unpredictable.
2. A thread that started earlier may or may not complete its work sooner than the thread that started later, even if it has less work to do.
3. Any attempt to control exact execution order will very likely impact performance and may result in all sorts of unwanted side effects, which will be discussed later in this chapter.

The following analogy can be helpful in understanding concurrent code execution:

Imagine that a CPU core is a road and a thread is a car. It would be rather inefficient to build a separate road for every car. Furthermore, one would expect a road to be shared by many cars, which helps to explain why multithreading makes much more efficient use of computer resources. However, even though a given road is shared among many cars, it does not mean that two cars can occupy the same spot on that road at the same time—that is quite obviously an undesirable situation. So cars have to take turns and yield to one another to be able to share the same road, very much like threads have to share a CPU core. It may feel like the cars are using a given road in parallel, but strictly speaking they are actually doing it concurrently. Finally, there is no guarantee that the car that started its journey earlier will get to where it wants to go sooner or later than another car, considering unpredictable circumstances such as being stuck in traffic. This is similar to a concurrent thread that may have to wait for a CPU time slice, if the CPU time has to be shared with many other threads and tasks on the computer. An attempt to control the exact execution order, to ensure that a certain thread gets priority, can be compared to stopping all traffic to yield to an ambulance or a police car—while they get priority, everyone else is stopped, causing overall performance to degrade. It may be tolerable for extraordinary cases, but it would be a traffic disaster if the policy for using the road was to prioritize each and every car on the road. This means that it is best to embrace the stochastic nature of concurrent code execution and try not to control the exact execution order, or at least only do it in exceptional cases.

22.2 Runtime Organization for Thread Execution

Most JVM implementations run as a single process, but allow multiple threads to be created. Runtime organization for thread execution in the JVM is depicted in **Figure 22.1**. The JVM has designated memory areas, called *runtime data areas*, that are deployed for various purposes during program execution. **Figure 22.1** shows data areas that are created specifically for each thread and are private to each thread. The figure also shows data areas that are shared by all threads, called *shared memory*.

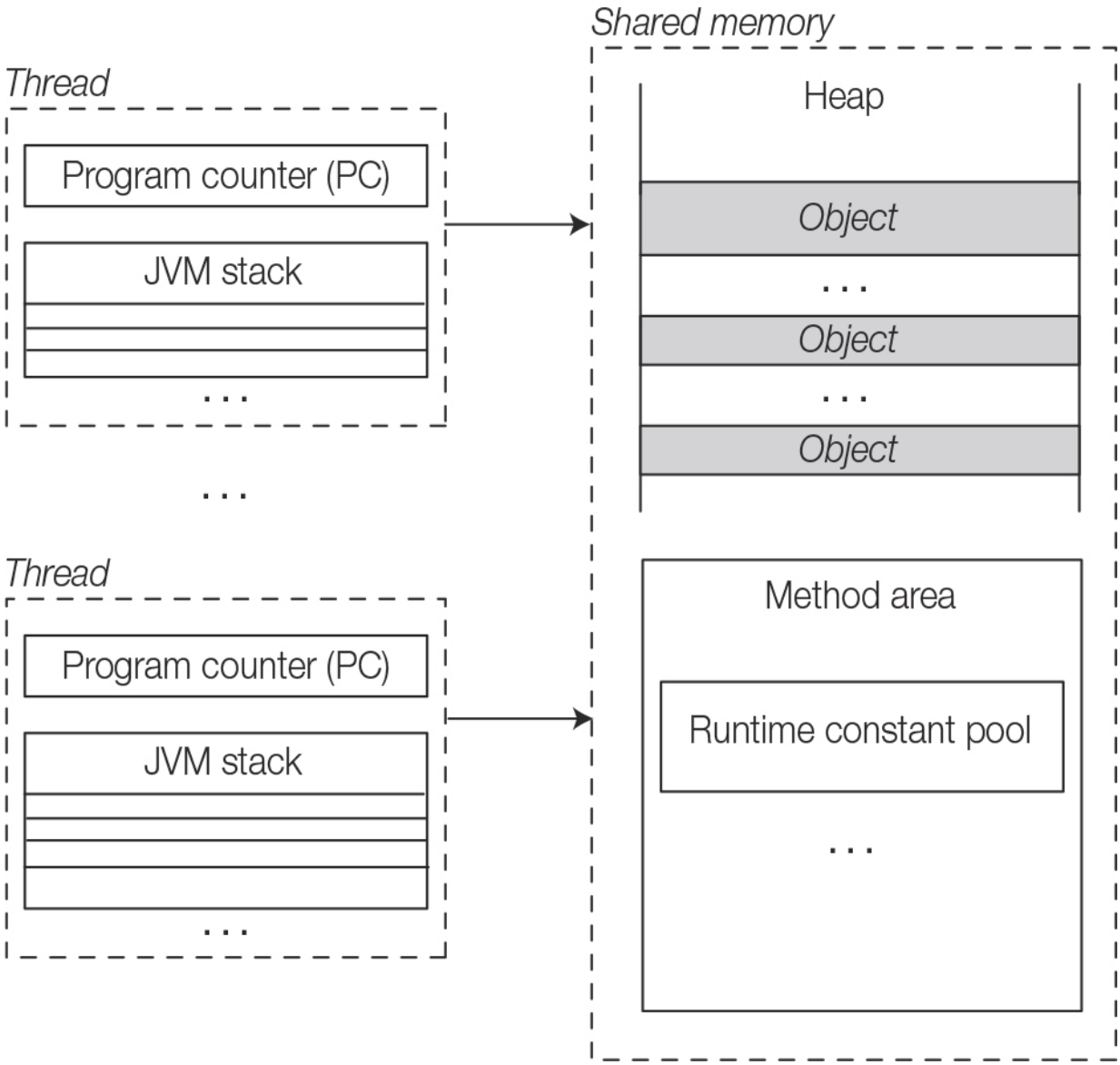

Figure 22.1 *Runtime Data Areas*

Each thread has the following data areas, which are private to the thread:

- *JVM stack*: A JVM stack (also known as *execution stack*, *call stack*, or *frame stack*) is created for each thread when the thread starts, and is used for bookkeeping method executions in the thread, as explained in §7.1, p. 365. This is also where all local variables for each active method invocation are stored. Note that each thread takes care of its own exception handling, and thus does not affect other threads.
- *Program counter* (PC): This register is created for each thread when the thread starts, and stores the address of the JVM instruction currently being executed.

The following data areas are shared by all threads:

- *Heap*: This shared memory space is where objects are created, stored, and garbage collected.
- *Method area*: This is created when the JVM starts. It stores the runtime constant pool, field and method information, static variables, and method bytecode for each of the classes and interfaces loaded by the JVM.
- *Runtime constant pool*: In addition to storing the constants defined in each class and interface, this also stores references to all methods and fields. The JVM uses the information in

this pool to find the actual address of a method or a field in memory.

Threads make the runtime environment asynchronous, allowing different tasks to be performed concurrently. Using this powerful paradigm in Java centers on understanding the following aspects of multithreaded programming:

- Creating threads and providing the code that gets executed by a thread (**p. 1370**)
- Understanding the thread lifecycle (**p. 1380**)
- Understanding thread issues that occur due to the execution model where concurrent threads are accessing shared memory (**p. 1408**)

22.3 Creating Threads

A thread in Java is represented by an object of the `java.lang.Thread` class. Every thread in Java is created and controlled by an object of this class. Often the thread (of execution) and its associated `Thread` object are thought of as being synonymous. Implementing the task performed by a thread is achieved in one of two ways:

- Implementing the functional interface `java.lang.Runnable`
- Extending the `java.lang.Thread` class

In both cases, thread creation and management is controlled by the application. **Chapter 23**, **p. 1419**, discusses high-level concurrency features that abstract the drudgery of thread creation and management, facilitating the building of massively concurrent applications.

Implementing the `Runnable` Interface

The `Runnable` functional interface has the following specification, comprising a single abstract method declaration:

```
@FunctionalInterface
public interface Runnable {
  void run();
}
```

A thread, which is created based on an object that implements the `Runnable` interface, will execute the code defined in the `public` method `run()`. In other words, the code in the `run()` method defines an independent path of execution and thereby the entry and the exits for the thread. A thread ends when the `run()` method ends, either by normal completion or by throwing an uncaught exception. Note that the method `run()` does not return a value, does not take any parameters, and does not throw any checked exceptions.

The procedure for creating threads based on the `Runnable` interface is as follows:

1. Implement the `Runnable` interface, providing the `run()` method that will be executed by the thread. This can be done by providing a concrete or an anonymous class that imple-

ments the `Runnable()` interface. Since the interface is a functional interface, it can also be implemented by a lambda expression, typically for simple tasks.
2. An object of the `Thread` class is created by passing the `Runnable` implementation from step 1 as an argument in the `Thread` constructor call. The `Thread` object now has a `Runnable` object that implements the `run()` method.
3. The `start()` method is invoked on the `Thread` object created in step 2. The `start()` method returns immediately after a thread has been spawned. In other words, the call to the `start()` method is asynchronous.

When the thread, represented by the `Thread` object on which the `start()` method was invoked, gets to run, it executes the `run()` method of the `Runnable` object. This sequence of events is illustrated in **Figure 22.2**.

In the following code, the functional interface `Runnable` is implemented by a lambda expression that is passed to the `Thread` object. The code will create a thread and start the thread. When the thread gets to run, it will execute the print statement.

Click here to view code image

```
Thread thread = new Thread(
    () -> System.out.println("Harmonious threads create beautiful applications.")
);
thread.start();
```

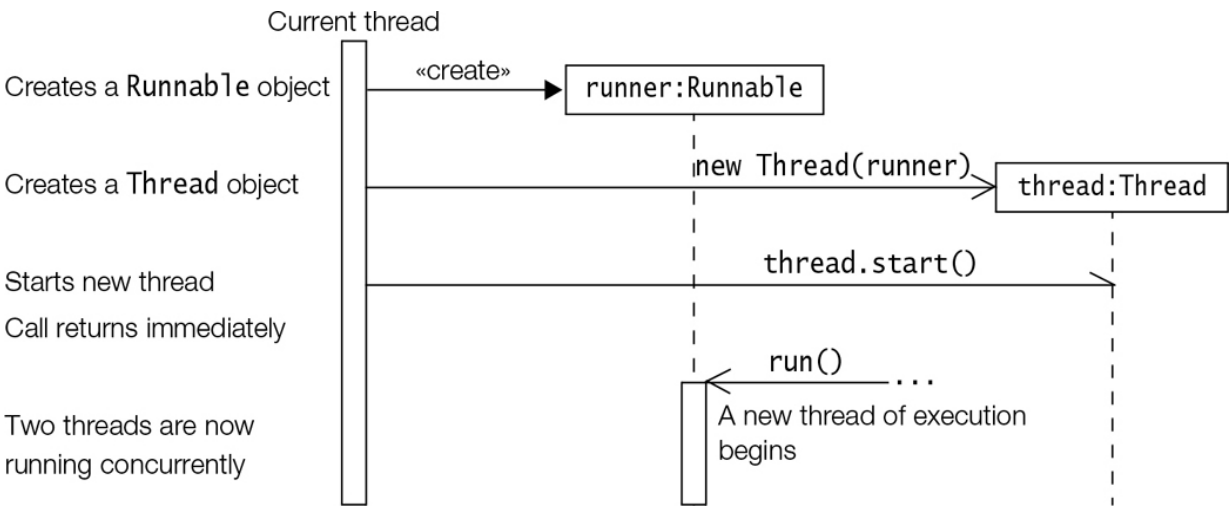

Figure 22.2 *Spawning Threads Using a* `Runnable` *Object*

The following is a summary of selected constructors and methods from the `java.lang.Thread` class:

Click here to view code image

```
Thread(Runnable threadTarget)
Thread(Runnable threadTarget, String threadName)
```

The argument `threadTarget` is the object whose `run()` method will be executed when the thread is started. The argument `threadName` can be specified to give an explicit name for the thread, rather than an automatically generated one.

```
void start()
```

Spawns a new thread—that is, the new thread will begin execution as a child thread of the current thread. The spawning is done asynchronously as the call to this method returns immediately. It throws an `IllegalThreadStateException` if the thread is already started or it has already completed execution.

```
void run()
```

The `Thread` class implements the `Runnable` interface by providing an implementation of the `run()` method. This implementation in the `Thread` class does nothing and returns. Subclasses of the `Thread` class should override this method. If the current thread is created using a separate `Runnable` object, the `run()` method of this `Runnable` object is called.

```
static Thread currentThread()
```

Returns a reference to the `Thread` object of the currently executing thread.

Click here to view code image

```
final String getName()
final void setName(String name)
```

The first method returns the name of the thread. The second one sets the thread's name to the specified argument.

Click here to view code image

```
final void setDaemon(boolean flag)
final boolean isDaemon()
```

The first method sets the status of the thread either as a daemon thread or as a normal thread (p. 1377), depending on whether the argument is `true` or `false`, respectively. The status should be set before the thread is started. The second method returns `true` if the thread is a daemon thread; otherwise, it returns `false`.

A slightly more elaborate example of creating a thread is presented in **Example 22.1**. The class `Counter` implements the `Runnable` interface. At (1), the class defines the `run()`

method that constitutes the code to be executed in a thread. The `while` loop in the `run()` method executes as long as the current value is less than 5. In each iteration of the `while` loop, the old value and the new incremented value of the counter is printed, as shown at (3). Also, in each iteration, the thread will sleep for 500 milliseconds, as shown at (4). While it is sleeping, other threads may run (p. 1395).

The static method `currentThread()` in the `Thread` class can be used to obtain a reference to the `Thread` object associated with the current thread. We can call the `get-Name()` method on the current thread to obtain its name. An example of its usage, shown at (2), prints the name of the thread executing the `run()` method. Another example of its usage, shown at (5), prints the name of the thread executing the `main()` method.

The `Client` class in **Example 22.1** uses the `Counter` class. It creates an object of the class `Counter` at (6). This `Counter` object is passed to two `Thread` objects at (7). The threads are started at (8). In other words, we have two threads that will increment the value in the same `Counter` object.

Both `Thread A` and `Thread B` are *child* threads of the *main* thread. They inherit the normal-thread status from the main thread (p. 1377). The output shows that the main thread finishes executing before the child threads. However, the program will continue running until the child threads have completed their execution of the `run()` method in the `Counter` object.

If a thread has been started or if it has completed, invoking the `start()` method again on the thread results in an `IllegalThreadStateException`, as shown at (9). Note that this does *not* terminate the thread that has already been started. If the thread has completed, a new thread must be created before the `start()` method can be called.

Since thread scheduling is not predictable (p. 1386) and **Example 22.1** does not enforce any synchronization between the two threads in accessing the current value in the `Counter` object, the output shown may vary. The output from the two child threads is interspersed. The output also shows that the counter was incremented by 2 when `Thread B` executed for the first time! This is an example of *thread interference*. The challenge in multithreaded programming is to synchronize how shared data is accessed by the threads in the application.

Example 22.1 *Implementing the* `Runnable` *Interface*

Click here to view code image

```
public class Counter implements Runnable {
  private int currentValue = 0;

  @Override
  public void run() {                          // (1) Thread entry point
    String threadName = Thread.currentThread().getName(); // (2)
    while (currentValue < 5) {
      System.out.printf("%s: old:%s new:%s%n",
                        threadName,            // (3) Print thread name,
```

```
                        this.currentValue,    //     old value,
                       ++this.currentValue);  //     new incremented value.
      try {
        Thread.sleep(500);                    // (4) Current thread sleeps.
      } catch (InterruptedException e) {
        System.out.println(threadName + " interrupted.");
      }
    }
    System.out.println("Exiting " + threadName);
  }
}
```

Click here to view code image

```
public class Client {
  public static void main(String[] args) {
    String threadName = Thread.currentThread().getName();  // (5) main thread
    System.out.println("Method main() runs in thread " + threadName);

    // Create a Counter object:                            // (6)
    Counter counter = new Counter();

    // Create two threads with the same Counter:           // (7)
    Thread threadA = new Thread(counter, "Thread A");
    Thread threadB = new Thread(counter, "Thread B");

    // Start the two threads:                              // (8)
    System.out.println("Starting " + threadA.getName());
    threadA.start();
//  threadA.start();                          // (9) IllegalThreadStateException
    System.out.println("Starting " + threadB.getName());
    threadB.start();

    System.out.println("Exiting Thread " + threadName);    // (10)
  }
}
```

Probable output from the program:

Click here to view code image

```
Method main() runs in thread main
Starting Thread A
Starting Thread B
Exiting Thread main
Thread B: old:0 new:2
Thread A: old:0 new:1
Thread B: old:2 new:3
Thread A: old:3 new:4
Thread B: old:4 new:5
```

```
Exiting Thread A
Exiting Thread B
```

Extending the `Thread` Class

A class can also extend the `Thread` class to create a thread. A typical procedure for doing this is as follows (see **Figure 22.3**):

1. A concrete or an anonymous class extending the `Thread` class overrides the `run()` method from the `Thread` class to define the code executed by the thread.
2. This subclass may call a `Thread` constructor explicitly in its constructors to initialize the thread, using the `super()` call.
3. The `start()` method inherited from the `Thread` class is invoked on an object of the class to make the thread eligible for running. This method should never be overridden.

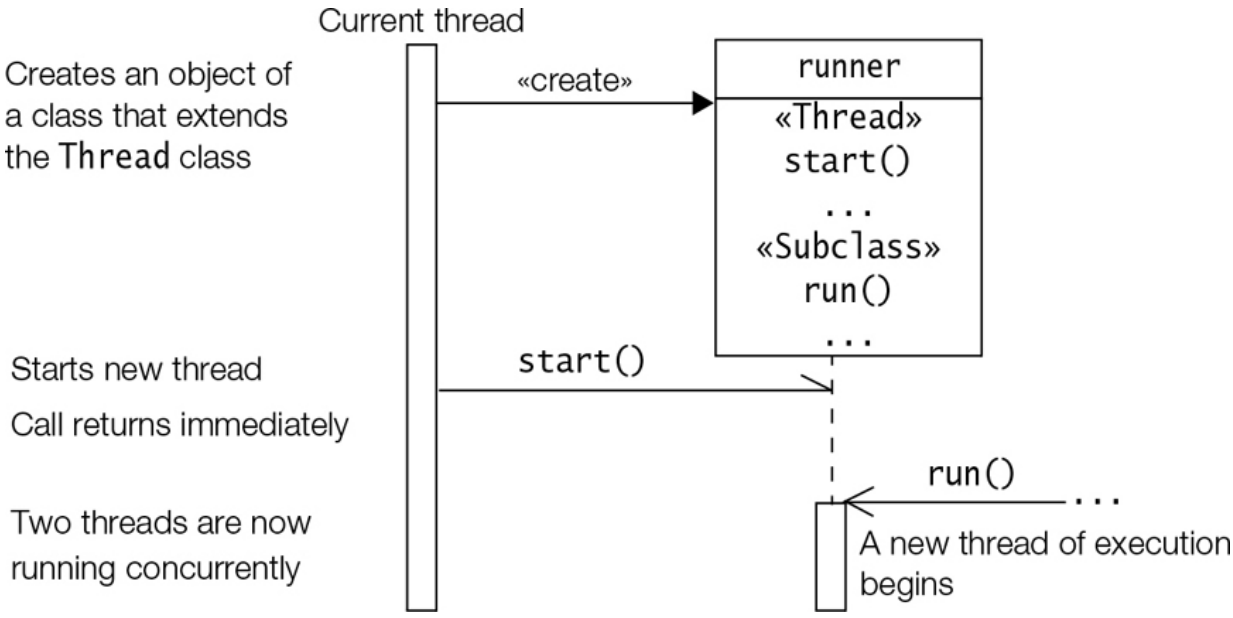

Figure 22.3 *Spawning Threads—Extending the* `Thread` *Class*

The simple example below shows an anonymous class that extends the `Thread` class and overrides the `run()` method from the `Thread` superclass. The code will create a thread and start the thread:

Click here to view code image

```
(new Thread() {
   @Override
   public void run() {
     System.out.println("Harmonious threads create beautiful applications.")
   }
 }
).start();
```

In **Example 22.2**, the `Counter` class from **Example 22.1** has been modified to illustrate creating a thread by extending the `Thread` class. The program output shows that the `Client` class

creates two threads and exits, but the program continues running until the child threads have completed execution. The two child threads are independent, each having its own counter and executing its own `run()` method. Again note that the main thread finished execution before the child threads, but the JVM does not terminate before the child threads are done.

The `Thread` class implements the `Runnable` interface, which means that this approach is not much different from implementing the `Runnable` interface directly. The main difference is that the roles of the `Runnable` object and the `Thread` object are combined in a single object by extending the `Thread` class.

In the previous two examples, the code to create the `Thread` object, to set the thread name, and to call the `start()` method to initiate the thread execution is in the client code. In **Example 22.2**, however, setting the name and starting the thread can be placed in the constructor of the `Counter` subclass, as it inherits from the `Thread` class—an exercise the studious reader is encouraged to undertake.

Example 22.2 *Extending the* `Thread` *Class*

Click here to view code image

```
public class Counter extends Thread {
  public int currentValue;
  public Counter() { this.currentValue = 0; }
  public int getValue() { return this.currentValue; }

  @Override
  public void run() {                          // (1) Override from superclass.
    while (this.currentValue < 5) {
      System.out.printf("%s: %s%n",
                  super.getName(),             // (2) Print thread name,
                  this.currentValue++);        //     current value, and increment.
      try {
        Thread.sleep(500);                     // (3) Current thread sleeps.
      } catch (InterruptedException e) {
        System.out.println(super.getName() + " interrupted.");
      }
    }
    System.out.println("Exiting " + super.getName());
  }
}
```

Click here to view code image

```
public class Client {
  public static void main(String[] args) {
    String threadName = Thread.currentThread().getName();              // (4)
    System.out.println("Method main() runs in thread " + threadName);
```

```
        // Create two Counter objects that extend the Thread class:          (5)
        Counter counterA = new Counter();
        Counter counterB = new Counter();

        // Set the names for the two threads:                                (6)
        counterA.setName("Counter A");
        counterB.setName("Counter B");
        // Mark the threads as daemon threads:                               (7)
//      counterA.setDaemon(true);
//      counterB.setDaemon(true);

        // Start the two threads:                                          // (8)
        System.out.println("Starting " + counterA.getName());
        counterA.start();
        System.out.println("Starting " + counterB.getName());
        counterB.start();

        System.out.println("Exiting " + threadName);
    }
}
```

Probable output from the program:

Click here to view code image

```
Method main() runs in thread main
Starting Counter A
Counter A: 0
Starting Counter B
Exiting main
Counter B: 0
Counter A: 1
Counter B: 1
Counter B: 2
Counter A: 2
Counter A: 3
Counter B: 3
Counter A: 4
Counter B: 4
Exiting Counter B
Exiting Counter A
```

When creating threads, there are a few reasons why implementing the `Runnable` interface may be preferable to extending the `Thread` class:

- Extending the `Thread` class means that the subclass cannot extend any other class, whereas a class implementing the `Runnable` interface has this option.
- Extending the `Thread` class also means that the task of the thread—that is, executing the `run()` method—cannot be shared by several threads. This does not mean that the threads

cannot access shared resources through their `run()` methods—which is typical in multi-threaded applications.
- A class might only be interested in being runnable, and therefore, inheriting the full overhead of the `Thread` class would be excessive.

Types of Threads

The runtime environment distinguishes between *normal threads* (also called *user threads*) and *daemon threads*. As long as a normal thread is alive—meaning that it has not completed executing its task—the JVM does not terminate. A daemon thread is at the mercy of the runtime system: It is stopped if no more normal threads are running, thus terminating the program. Daemon threads exist only to serve normal threads. It is not a good strategy to run any clean-up code in daemon threads, which might not get executed if a daemon thread is stopped.

Uncommenting the statements at (7) in **Example 22.2**:

Click here to view code image

```
counterA.setDaemon(true);
counterB.setDaemon(true);
```

illustrates the daemon nature of threads. The program execution will now terminate after the main thread has completed, without waiting for the daemon `Counter` threads to finish normally:

Click here to view code image

```
Method main() runs in thread main
Starting Counter A
Starting Counter B
Counter A: 0
Counter B: 0
Exiting main
```

When a standalone application is run, a normal thread is automatically created to execute the `main()` method of the application. This thread is called the *main thread*. If no other normal threads are spawned, the program terminates when the `main()` method finishes executing. All other threads, called *child* threads, are spawned from the main thread, inheriting its normal-thread status. The `main()` method can then finish, but the program will keep running until all normal threads have completed. The status of a spawned thread can be set as either daemon or normal, but this must be done before the thread is *started*. Any attempt to change the status after the thread has been started throws an unchecked `IllegalThreadState-Exception`. A child thread inherits the thread status of its parent thread.

When a GUI application is started, a special thread is automatically created to monitor the user–GUI interaction. This normal thread keeps the program running, allowing interaction between the user and the GUI, even though the main thread might have completed after the `main()` method finished executing.

Review Questions

22.1 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
public class MyClass extends Thread {
  public MyClass(String s) { msg = s; }
  String msg;
  public void run() {
    System.out.println(msg);
  }
  public static void main(String[] args) {
    new MyClass("Hello");
    new MyClass("World");
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile without errors and will print `Hello` and `World`, in that order, every time it is run.

c. The program will compile without errors, and will print a never-ending stream of `Hello` and `World`.

d. The program will compile without errors, and will print `Hello` and `World` when run, but the order is unpredictable.

e. The program will compile without errors, and will simply terminate without any output when run.

22.2 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
class R1 implements Runnable {
  public void run() {
    System.out.print(Thread.currentThread().getName());
```

```
  }
}
public class R2 implements Runnable {
  public void run() {
    new Thread(new R1(),"|R1a|").run();
    new Thread(new R1(),"|R1b|").start();
    System.out.print(Thread.currentThread().getName());
  }

  public static void main(String[] args) {
    new Thread(new R2(),"|R2|").start();
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile without errors, and will print `|R1a|` twice and `|R2|` once, in some order, every time it is run.

c. The program will compile without errors, and will print `|R1b|` twice and `|R2|` once, in some order, every time it is run.

d. The program will compile without errors, and will print `|R1b|` once and `|R2|` twice, in some order, every time it is run.

e. The program will compile without errors, and will print `|R1a|` once, `|R1b|` once, and `|R2|` once, in some order, every time it is run.

22.3 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
public class Threader extends Thread {
  Threader(String name) {
    super(name);
  }
  public void run() throws IllegalThreadStateException {
    System.out.println(Thread.currentThread().getName());
    throw new IllegalThreadStateException();
  }

  public static void main(String[] args) {
    new Threader("|T1|").start();
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile without errors, will print `|T1|`, and will terminate normally every time it is run.

c. The program will compile without errors, will print `|T1|`, and will throw an unchecked `IllegalThreadStateException` every time it is run.

d. None of the above

22.4 Thread Lifecycle

Before diving into the lifecycle of a thread, a few important thread-related concepts should be understood.

Objects, Monitors, and Locks

In Java, *each* object has a *monitor* associated with it—including arrays. A thread can *lock* and *unlock* a monitor, but only one thread at a time can *acquire* the lock on a monitor. The thread that acquires the lock is said to *own* the lock and has exclusive access to the object whose monitor was locked. Any other threads trying to acquire the lock are *blocked* and placed in the *entry set* of the monitor until the lock is *released*. At that time, if the entry set is not empty, a blocked thread is allowed to acquire the lock at the discretion of the JVM, based on the thread scheduling policy in effect. The locking mechanism thus implements *mutual exclusion* (also known as *mutex*).

It should be made clear that programs should not make any assumptions about the order in which threads are granted ownership of a lock while waiting in the entry set of the monitor providing the lock. Thread state transitions for lock acquisition are explained later (**p. 1388**).

The locking mechanism on a monitor is referred to by various names in the literature: *intrinsic lock*, *monitor lock*, *object-level lock*, *monitor*, or just *lock*. Conceptually, the lock is on the object, and therefore we will refer to this locking mechanism provided by a monitor associated with an object as the *object lock* or just *lock*, when it is clear which lock is being referred to.

Classes also have a *class lock* (a.k.a. *class-level lock*) that is analogous to the object lock. Such a lock is actually a lock on the `java.lang.Class` object that represents the class at runtime in the JVM. Given a class `A`, the reference `A.class` denotes this unique `Class` object. The class lock can be used in much the same way as an object lock to implement mutual exclusion.

The following `static` method of the `Thread` class can be used to determine whether the current thread holds a lock on a specific object:

> **Click here to view code image**

```
static boolean holdsLock(Object obj)
```

Thread States

Understanding the lifecycle of a thread can be beneficial when programming with threads. Threads can exist in different states. Just because the `start()` method has been called on the thread, it does not mean that the thread has access to the CPU and can start executing straightaway. Several factors determine how it will proceed.

Figure 22.4 shows the states and the transitions in the lifecycle of a thread. The figure shows the main transitions that occur during the lifecycle of a thread—in particular, transitions pertaining to *blocked I/O* (**p. 1405**) and *thread interruption* (**p. 1393**) have been omitted.

The thread states are represented by enum constants and summarized in **Table 22.1**. The `Thread` class provides the `getState()` method to determine the state of the current thread. The method returns a constant of type `Thread.State` (i.e., the type `State` is a static inner `enum` type declared in the `Thread` class).

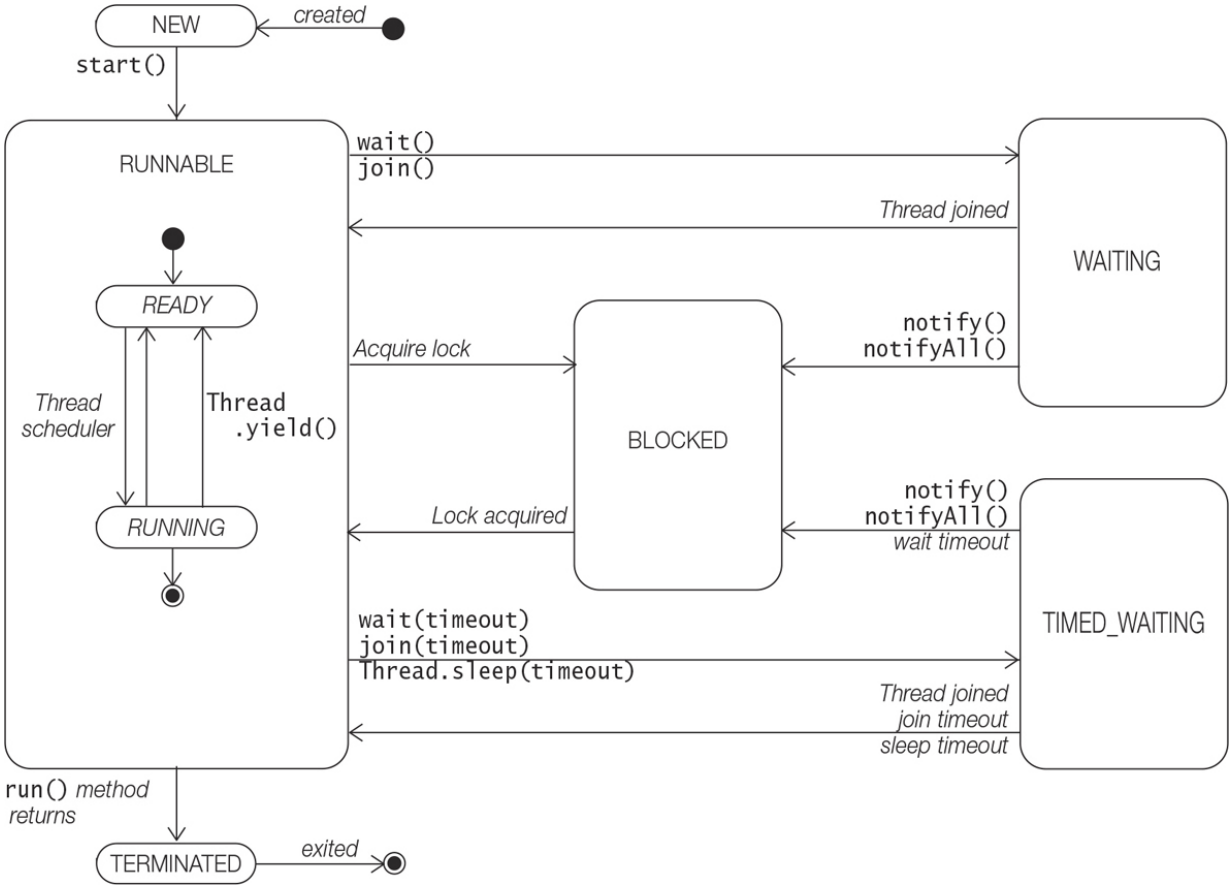

Figure 22.4 *Thread States*

Table 22.1 *Thread States Defined by the* `Thread.State` *Enum Type*

Constant in the `Thread.State` enum type (see Figure 22.4)	Description of the state
`NEW`	A thread in this state has been created, but it has not yet started. A thread is started by calling its `start()` method (p. 1370).
`RUNNABLE`	The `RUNNABLE` states can be characterized by two *substates* which are not observable: • *READY substate*: A thread starts life in the *READY* substate (p. 1386), and also when it transitions from a *non-runnable state* to the `RUNNABLE` state. • *RUNNING substate*: If a thread is in the *RUNNING* substate, it means that the thread is currently executing (p. 1386), and is deemed the *running thread*. From the *RUNNING* substate, a thread can transition either to the *READY* substate, to a non-runnable state, or to the `TERMINATED` state.
`BLOCKED`	A thread is blocked while waiting to acquire a lock (p. 1380, p. 1388). A thread is blocked while waiting for I/O (p. 1405).
`WAITING`	A thread is in this state indefinitely for the following reasons, until the action it is waiting for occurs: • A thread is waiting indefinitely for join completion: The thread awaits completion of another thread (p. 1402). • A thread is waiting indefinitely for notification: The thread awaits notification from another thread (p. 1396).
`TIMED_WAITING`	A thread is in this state for at least a *specified amount of time* for the following reasons, unless the action it is waiting for occurs before timeout: • Timed waiting for join completion: The thread awaits completion of another thread (p. 1402). • Timed waiting for notification: The thread awaits notification from another thread (p. 1396). • Timed waiting to wake up from sleep (p. 1395).

Constant in the `Thread.State` enum type (see Figure 22.4)	Description
`TERMINATED`	The `run()` method of the thread has completed execution or terminated. Once in this state, the thread can never run again (p. 1405).

The rest of this section will expound on the thread states and the transitions between them during the lifecycle of a thread. However, the following remarks on the thread lifecycle in Figure 22.4 should be noted.

- The `RUNNABLE` state is a *compound state* having two *non-observable substates*: *READY* and *RUNNING*. Being *non-observable* means it is not possible to distinguish which substate the thread is in once it is in the `RUNNABLE` state.
- The following states are characterized as *non-runnable* states: `BLOCKED`, `WAITING`, and `TIMED_WAITING`. A *running thread*—one in the *RUNNING* substate—can transit to one of the non-runnable states, depending on the circumstances. A thread remains in a non-runnable state until a specific transition occurs. A thread does not go directly to the *RUNNING* substate from a non-runnable state, but transits first to the *READY* substate.

Selected methods from the `Thread` class are presented below. Examples of their usage are presented in subsequent sections.

```
final boolean isAlive()
```

This method can be used to find out if a thread is alive or dead. A thread is *alive* if it has been started but not yet terminated—that is, it is not in the `TERMINATED` state.

```
Thread.State getState()
```

Returns the state of this thread (Table 22.1). It should be used for monitoring the state and not for synchronizing control.

Click here to view code image

```
final int getPriority()
final void setPriority(int newPriority)
```

The first method returns the priority of a thread (p. 1385). The second method changes its priority. The priority set will be the minimum of the specified `newPriority` and the maximum priority permitted for this thread. There is no guarantee that a thread with a higher priority will be chosen to run.

```
static void yield()
```

Causes the current thread to temporarily pause its execution and, thereby, allow other threads to execute. It is up to the JVM to decide if and when this transition will take place. The transition is from the *RUNNING* substate to the *READY* substate in the `RUNNABLE` state (**Figure 22.4**).

Click here to view code image

```
static void sleep(long millisec) throws InterruptedException
static void sleep(long millis, int nanos) throws InterruptedException
```

The current thread sleeps for at least the specified time before it becomes eligible for running again. The transition is from the *RUNNING* substate of the `RUNNABLE` state to the `TIMED_WAITING` state (**Figure 22.4**).

Click here to view code image

```
final void join() throws InterruptedException
final void join(long millisec) throws InterruptedException
```

A call to any of these two methods invoked on a thread will wait and not return until either the thread has completed or it is timed out after the specified time, respectively. In **Figure 22.4**, the transition is from the *RUNNING* substate of the `RUNNABLE` state to the `WAITING` state for the first method and to the `TIMED_WAITING` state for the second method.

Click here to view code image

```
static Map<Thread, StackTraceElement[]> getAllStackTraces()
```

Returns a map of stack traces for all alive threads. Keep in mind that each thread has its own JVM stack for method execution. The map can be used, for example, to extract a set of all alive threads.

Example 22.3 illustrates transitions between thread states. A thread at (1) sleeps a little at (2) and then does some computation in a loop at (3), after which the thread terminates. The `main()` method monitors the thread in a loop at (4), printing the thread state returned by the `getState()` method. The output shows that the thread starts off in the `NEW` state and goes through the `RUNNABLE` state when the `run()` method starts to execute, and then transits to the `TIMED_WAITING` state to sleep. On waking up, it computes the loop in the `RUNNABLE` state, and transits to the `TERMINATED` state when the `run()` method completes execution.

Example 22.3 *Thread States*

Click here to view code image

```
public class ThreadStates {

  private static Thread t1 = new Thread("T1") {    // (1)
    @Override public void run() {
      try {
        sleep(1);                                  // (2)
        for (int i = 10000; i > 0; i--) {;}        // (3)
      } catch (InterruptedException ie) {
        ie.printStackTrace();
      }
    }
  };

  public static void main(String[] args) {
    System.out.println(t1.getState());              // (4)
    t1.start();
    while (true) {                                  // (5)
      Thread.State state = t1.getState();
      System.out.println(state);
      if (state == Thread.State.TERMINATED) {
        break;
      }
    }
  }
}
```

Probable output from the program:

```
NEW
RUNNABLE
TIMED_WAITING
...
TIMED_WAITING
RUNNABLE
...
RUNNABLE
TERMINATED
```

Thread Priorities

Threads are assigned priorities that the thread scheduler *can* use to determine how the threads will be scheduled. The thread scheduler can use thread priorities to determine which thread gets to run. The thread scheduler favors giving CPU time to the thread with the highest priority in the *READY* substate. This is not necessarily the thread that has been in the *READY* substate the longest time. Heavy reliance on thread priorities for the behavior of a program can make the program unportable across platforms, as thread scheduling is host platform–dependent.

Selective priorities are defined by `enum` constants in the `Thread` class, shown in **Table 22.2**.

Table 22.2 *Selective Priorities Defined by the* `Thread` *Class*

Enum type `java.lang.Thread`	Value	Description
`MIN_PRIORITY`	`0`	Lowest priority
`NORM_PRIORITY`	`5`	Default priority
`MAX_PRIORITY`	`10`	Highest priority

A thread inherits the priority of its parent thread. The priority of a thread can be set using the `setPriority()` method and read using the `getPriority()` method, both of which are defined in the `Thread` class. The following code sets the priority of the thread `myThread` to the minimum of two values: maximum priority and current priority incremented to the next level. It also shows the default text representation of a thread, `[thread_name, priority, parent_thread_name]`.

Click here to view code image

```
Thread myThread = new Thread(() ->
    System.out.println(Thread.currentThread() + ": Don't mess with my priority!")
);
System.out.println(myThread);
myThread.setPriority(Math.min(Thread.MAX_PRIORITY, myThread.getPriority() + 1));
myThread.start();
```

Output from the code:

Click here to view code image

```
Thread[Thread-0,5,main]
Thread[Thread-0,6,main]: Don't mess with my priority!
```

The `setPriority()` method is an *advisory* method, meaning that it provides a hint from the program to the JVM, which the JVM is in no way obliged to honor. A thread with higher priority is not guaranteed to complete its work faster than a thread with lower priority. The method can be used to fine-tune the *performance* of the program, but should not be relied upon for the *correctness* of the program.

Thread Scheduler

Schedulers in JVM implementations usually employ one of the following two strategies, which come into play in the `RUNNABLE` state:

- Preemptive scheduling
  If a thread with a higher priority than the current running thread moves to the *READY* substate, the current running thread can be *preempted* (moved from the *RUNNING* substate to the *READY* substate) to let the higher-priority thread execute.
- Time-sliced or round-robin scheduling
  A running thread is allowed to execute for a fixed length of time in the *RUNNING* substate, after which it moves to the *READY* substate to await its turn to run again.

It should be emphasized that thread schedulers are implementation- and platform-dependent; therefore, how threads will be scheduled is unpredictable, at least from platform to platform.

Starting a Thread

After a thread has been created, the thread is in the `NEW` state (**Figure 22.5**). Only after its `start()` method has been called, can the thread do useful work. This method is asynchronous—that is, it returns immediately. The newly created thread transits to the *READY* substate, waiting its turn to execute on the CPU at the discretion of the thread scheduler. The `getState()` method on this thread will return the value `Thread.State.RUNNABLE`. For a thread that is created but not started, the `getState()` method will return the value `Thread.State.NEW`. Details of creating and starting a thread were covered in an earlier section (**p. 1370**).

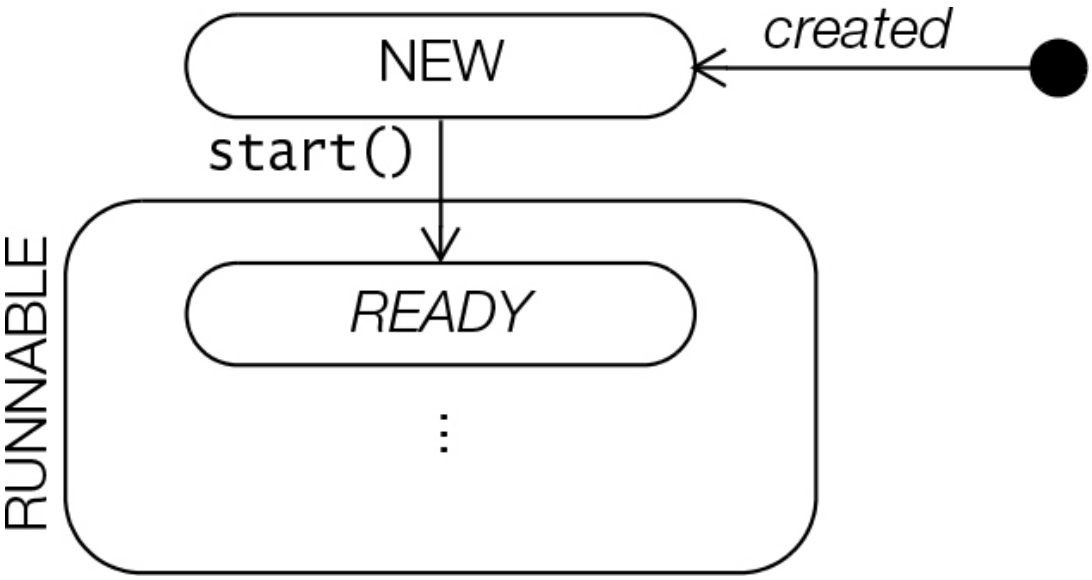

Figure 22.5 *Starting a Thread*

Running and Yielding

Once in the *READY* substate, the thread is eligible for running—that is, it waits for its turn to get CPU time. The thread scheduler decides which thread runs and for how long. **Figure 22.6** illustrates the transitions between the *READY* and *RUNNING* substates. The thread transits from the *READY* substate to the *RUNNING* substate when it is its turn to run. The `getState()` method on this thread will return the value `Thread.State.RUNNABLE`.

A call to the static method `yield()`, defined in the `Thread` class, may cause the current thread in the *RUNNING* substate to transit to the *READY* substate. If this happens, the thread is then at the mercy of the thread scheduler as to when it will run again. It is possible that if there are no threads in the *READY* substate, this thread can continue executing. If there are other threads in the *READY* substate, their priorities can influence which thread gets to execute.

As with the `setPriority()` method, the `yield()` method is also an advisory method, and therefore comes with no guarantees that the JVM will honor the call. A call to the `yield()` method does *not* affect any locks that the thread might hold.

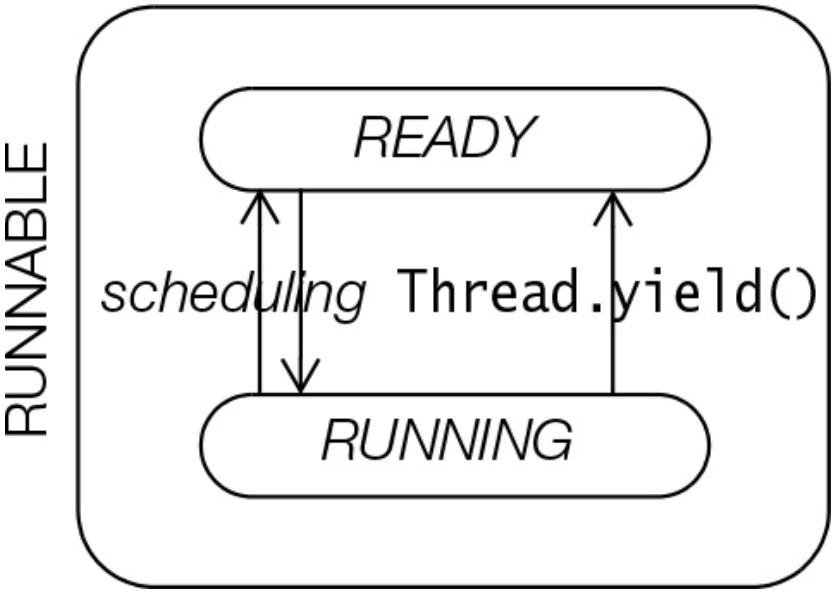

Figure 22.6 *Running and Yielding*

By calling the static method `yield()`, the running thread gives other threads in the *READY* substate a chance to run. A typical example where this can be useful is when a user has given some command to start a CPU-intensive computation, and has the option of cancelling it by clicking a CANCEL button. If the computation thread hogs the CPU and the user clicks the CANCEL button, chances are that it might take awhile before the thread monitoring the user input gets a chance to run and take appropriate action to stop the computation. A thread running such a computation should do the computation in increments, yielding between increments to allow other threads to run. This is illustrated by the following `run()` method:

Click here to view code image

```
public void run() {
  try {
    while (!done()) {
      doLittleBitMore();
      Thread.yield();                 // Current thread yields.
    }
  } catch (InterruptedException ie) { // Clean up if the thread is interrupted.
    doCleaningUp();
  }
}
```

Executing Synchronized Code

Threads share the same memory space—that is, they can share resources. However, there are critical situations where it is desirable that only one thread at a time has access to a shared resource. For example, crediting and debiting a shared bank account concurrently among several users without proper discipline will jeopardize the integrity of the account data. Java provides the concept of *synchronization* in order to control access to shared resources. It is one of the mechanisms that Java provides to write *thread-safe code* (§23.4, p. 1451).

The keyword `synchronized` and the intrinsic lock mechanism (p. 1380) form the basis for implementing synchronized execution of code. We refer to this as *intrinsic locking* (also referred to as *built-in synchronization*). There are two ways in which execution of code can be synchronized: by declaring `synchronized` *methods* or by using `synchronized` *statements*. The code that is synchronized to execute by one thread at a time is often referred to as a *critical section*.

Acquiring the Object Lock

In order to execute `synchronized` code, a thread must first acquire the lock of the relevant object, as illustrated in Figure 22.7. If this lock is not available, the thread transits from the *RUNNING* substate to the `BLOCKED` state. The thread is put in the *entry set* of the object. Threads in the `BLOCKED` state are grouped according to the object whose lock they are waiting for. The `getState()` method on a blocked thread in this state will return the value `Thread.State.BLOCKED`.

If there are several threads waiting for the same object lock, one is chosen at the discretion of the thread scheduler. It then transits to the *READY* substate where it takes its turn to run again.

Example 22.4 illustrates acquiring the lock in order to synchronize on a shared resource.

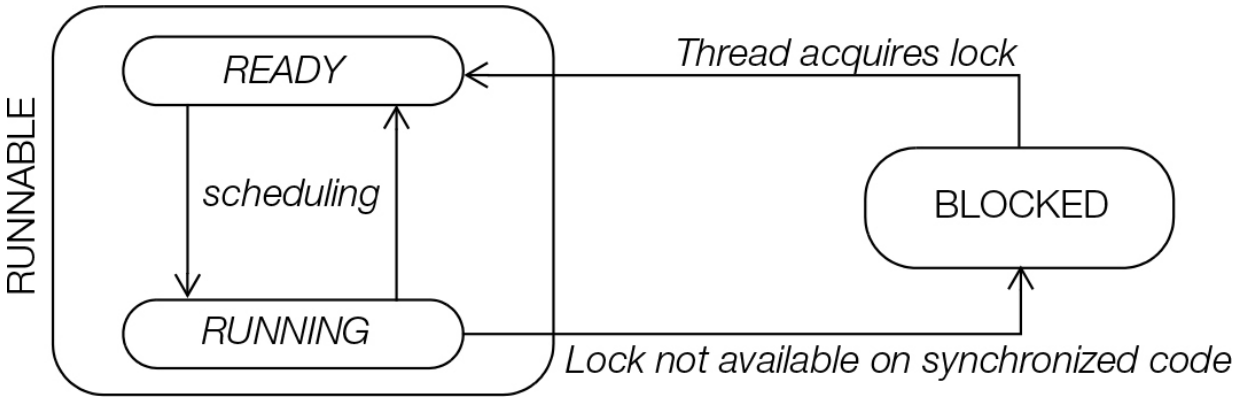

Figure 22.7 *Blocked for Lock Acquisition*

Synchronized Methods

If the methods of an object should only be executed by one thread at a time, then the declaration of all such methods should be specified with the keyword `synchronized`. A thread wish-

ing to execute a `synchronized` method must first acquire the object lock before it can execute the method on the object. This is simply achieved by calling the method. If the lock is already held by another thread, the calling thread waits in the entry set of the object lock, as explained earlier. No particular action on the part of the program is necessary to acquire the lock. A thread releases the lock on method return, regardless of whether the `synchronized` method completed execution normally or threw an uncaught exception. In both cases, if there are any blocked threads in the entry set, one is chosen to acquire the lock. In case of an uncaught exception, the exception is propagated through the JVM stack of the thread that released the lock.

Synchronized methods are useful in situations where methods can manipulate the state of a shared object in ways that can corrupt the state if executed concurrently.

Example 22.4 is a reworking of the counter implementation from **Example 22.1**. The `CounterX` class defines the `synchronized` method `increment()` at (1a) that prints the old value and the new incremented value in the counter.

The `main()` method in the `SynchronizedMethodDemo` class creates a `CounterX` object at (2) and a `Runnable` at (3) that calls the `increment()` method five times on this `CounterX` object. Two threads are created and started at (4) and (5) that will each call the synchronized `increment()` method on the *same* `CounterX` object five times—in contrast to **Example 22.1**, where each thread had its own counter.

From the output shown in **Example 22.4**, we can see that the main thread exits right after creating and starting the threads. The output shows that the two threads run mutually exclusive of each other. `Thread-0` increments the counter to 5 and `Thread-1` increments it further to 10.

It is instructive to run **Example 22.4** with (1a) commented out and (1b) uncommented. We see from the output that the execution of the threads is interleaved, and some results look dubious (`Thread-1: old:0 new:2`). Non-`synchronized` incrementing of the value in the counter between the two threads is a disaster waiting to happen. This is an example of what is called a *race condition* (also known as *thread interference*). It occurs when two or more threads simultaneously update a shared value and, due to the scheduling of the threads, can leave the value in an undefined or inconsistent state.

Running the program in **Example 22.4** with the `synchronized` version of the `increment()` at (1a) avoids any race conditions. The lock is only released when the `synchronized` method exits, guaranteeing a mutually exclusive increment of the counter by each thread.

Example 22.4 *Mutual Exclusion*

Click here to view code image

```
class CounterX {
  private int counter = 0;
```

```
  public synchronized void increment() {                          // (1a)
//public void increment() {                                       // (1b)
    System.out.println(Thread.currentThread().getName()
                       + ": old:" + counter + " new:" + ++counter);
  }
}
```

```
public class SynchronizedMethodDemo {
  public static void main(String[] args) {
    CounterX counter = new CounterX();                            // (2)
    Runnable r = () -> {                                          // (3)
      for (int i = 0; i < 5; i++) {
        counter.increment();
      }

      System.out.println("Exiting " + Thread.currentThread().getName());
    };
    new Thread(r).start();                                        // (4)
    new Thread(r).start();                                        // (5)
    System.out.println("Exiting thread " + Thread.currentThread().getName());
  }
}
```

Probable output from the program when run with (1a):

```
Exiting thread main
Thread-0: old:0 new:1
Thread-0: old:1 new:2
Thread-0: old:2 new:3
Thread-0: old:3 new:4
Thread-0: old:4 new:5
Exiting Thread-0
Thread-1: old:5 new:6
Thread-1: old:6 new:7
Thread-1: old:7 new:8
Thread-1: old:8 new:9
Thread-1: old:9 new:10
Exiting Thread-1
```

Probable output from the program when run with (1b):

```
Thread-1: old:0 new:2
Thread-1: old:2 new:3
Exiting thread main
Thread-0: old:0 new:1
Thread-1: old:3 new:4
Thread-0: old:4 new:5
Thread-0: old:6 new:7
```

```
Thread-1: old:5 new:6
Thread-1: old:8 new:9
Exiting Thread-1
Thread-0: old:7 new:8
Thread-0: old:9 new:10
Exiting Thread-0
```

Reentrant Synchronization

While a thread is inside a `synchronized` method of an object, all other threads that wish to execute this `synchronized` method or any other `synchronized` method of the object will have to wait in the entry set. This restriction does not apply to the thread that already has the lock and is executing a `synchronized` method of the object. Such a thread can invoke, directly or indirectly, other `synchronized` methods of the object without being blocked—that is, once a thread has a lock on an object, it can acquire the lock on the same object several times, called *reentrant synchronization*. The non-`synchronized` methods of the object can always be called at any time by any thread.

Examples of reentrant synchronization can be found in §23.4, p. 1460.

Synchronization on Class Lock

Static methods synchronize on the *class lock*. Acquiring and releasing a class lock by a thread in order to execute a `static synchronized` method is analogous to that of an object lock for a `synchronized` instance method. A thread acquires the class lock before it can proceed with the execution of any static `synchronized` method in the class, blocking other threads wishing to execute any `static synchronized` methods in the same class. The blocked threads wait in the entry set of the class lock. This does not apply to static, non-`synchronized` methods, which can be invoked by any thread. A thread acquiring the lock of a class to execute a `static synchronized` method has no effect on any thread acquiring the lock on any object of the class to execute a `synchronized` instance method. In other words, synchronization of static methods in a class is independent from the synchronization of instance methods on objects of the class.

A subclass decides whether the new definition of an inherited `synchronized` method will remain `synchronized` in the subclass.

Synchronized Statements

Whereas execution of `synchronized` methods of an object is synchronized on the lock of the object, the `synchronized` statement allows execution of arbitrary code to be synchronized on the lock of an *arbitrary object*. The general form of the `synchronized` statement (also called *synchronized block*) is as follows:

Click here to view code image

```
synchronized (object_reference_expression) { code_block }
```

The *object reference expression* must evaluate to a non-null reference value; otherwise, a `NullPointerException` is thrown. The *code block* is usually related to the object whose lock is acquired. This is analogous to a `synchronized` method, where the execution of the method is synchronized on the lock of the current object. The following code is equivalent to the `synchronized increment()` method in the `CounterX` class in **Example 22.4**:

Click here to view code image

```
public void increment() {
  synchronized(this) {                              // Synchronized statement
    System.out.println(Thread.currentThread().getName()
        + ": current: " + counter + " new:" + ++counter);
  }
}
```

Once a thread has entered the code block after acquiring the lock on the specified object, no other thread will be able to execute the code block, or any other code requiring the same object lock, until the lock is released. This happens when the execution of the code block completes normally or an uncaught exception is thrown. In contrast to `synchronized` methods, this mechanism allows fine-grained synchronization of code on arbitrary objects.

The object reference expression in the `synchronized` statement is mandatory. A class can choose to synchronize the execution of a part of a method by using the `this` reference and putting the relevant code of the method in the `synchronized` statement.

The curly brackets of the block cannot be omitted, even if the code block has just one statement.

Click here to view code image

```
class SmartClient {
  BankAccount account;
  // ...
  public void updateTransaction() {
    synchronized (account) {       // (1) synchronized statement
      account.update();            // (2)
    }
  }
}
```

In the example above, the code at (2) in the `synchronized` statement at (1) is synchronized on the `BankAccount` object. If several threads were to concurrently execute the method `updateTransaction()` on an object of `SmartClient`, the statement at (2) would be executed by

one thread at a time only after synchronizing on the `BankAccount` object associated with this particular instance of `SmartClient`.

Inner classes can access data in their enclosing context (§9.1, p. 491). An inner object might need to synchronize on its associated outer object in order to ensure integrity of data in the latter. This is illustrated in the following code where the `synchronized` statement at (5) uses the special form of the `this` reference to synchronize on the outer object associated with an object of the inner class. This setup ensures that a thread executing the method `incr()` in an inner object can only access the `private double` field `count` at (2) in the `synchronized` statement at (5) by first acquiring the lock on the associated outer object. If another thread has the lock of the associated outer object, the thread in the inner object has to wait for the lock to be released before it can proceed with the execution of the `synchronized` statement at (5). However, synchronizing on an inner object and on its associated outer object are independent of each other, unless enforced explicitly, as in the following code:

Click here to view code image

```
class Outer {                       // (1) Top-level Class
  private double count;             // (2)
  protected class Inner {           // (3) Non-static member Class
    public void incr() {            // (4)
      synchronized(Outer.this) {    // (5) Synchronized statement on Outer object
        ++count;                    // (6)
      }
    }
  }
}
```

A `synchronized` statement can also be specified on a class lock:

Click here to view code image

```
synchronized (<class_name>.class) { <code_block>}
```

The statement synchronizes on the lock of the object denoted by the reference *`class_name`*`.class`. This object (of type `Class`) represents the class at runtime. A `static synchronized` method `classAction()` in class `A` is equivalent to the following declaration:

Click here to view code image

```
static void classAction() {
  synchronized (A.class) {       // Synchronized statement on class A
    // ...
  }
}
```

In summary, a thread can acquire a lock by carrying out the following actions:

- By executing a `synchronized` instance method of an object of a class.
- By executing a `static synchronized` method of a class (in which case, the object is the `Class` object representing the class in the JVM).
- By executing the body of a `synchronized` statement that synchronizes either on an object of a class, or on the `Class` object representing a class in the JVM.
- Intrinsic locking does not provide any guarantees as to which thread among the waiting threads will acquire the intrinsic lock.
- Intrinsic lock acquisition is rigid: Either the thread acquires the intrinsic lock immediately if it is available, or the thread has to wait its turn among the waiting threads. There is no other lock acquisition policy—for example, timed waiting.
- An intrinsic lock is acquired and released in the same synchronized code—for example, in a synchronized method. It cannot be acquired or released in separate methods.

Interrupt Handling

The purpose of interrupts is to allow threads to inform each other when the task they are running might need attention. For example, a thread might be hoarding crucial resources or might have become unresponsive. Whatever corrective action is required is then at the discretion of the thread that is interrupted. Simply catching and ignoring an `InterruptedException` is not recommended.

The following selected methods from the `Thread` class can be used for handling interrupts in threads.

```
void interrupt()
```

Interrupts the thread on which it is invoked.

If the thread is in a non-runnable state, either blocked or waiting, the *interrupt status* of the thread is *cleared* and the thread will receive an `InterruptedException` when it gets to run.

For a thread that is in the `RUNNABLE` state, the *interrupt status* of the thread will be *set*.

```
boolean isInterrupted()
```

Checks whether this thread has been interrupted. The *interrupt status* of the thread is *not* affected.

```
static boolean interrupted()
```

Checks whether the current thread has been interrupted. The *interrupt status* of the thread is cleared.

A thread can be interrupted by invoking the method `interrupt()`. A thread can also invoke this method on itself. How the interrupt manifests in the interrupted thread depends on the state the thread is in when the `interrupt()` method is invoked. **Example 22.5** illustrates the two main scenarios that show how a thread can discover if it has been interrupted—and which a thread should be prepared to handle.

In **Example 22.5**, the `main()` method creates and starts a thread at (1) that executes the task defined at (3). The main thread uses an infinite loop at (2) in which it tries to send an interrupt to the `worker` thread by calling the `interrupt()` method. Sending an interrupt is dependent on the condition in the `if` statement being a random even number. If it cannot send an interrupt, the main thread sleeps for awhile before continuing to execute the infinite loop. The main thread exits the infinite loop (and terminates) when it has sent an interrupt to the `worker` thread.

The task executed by the `worker` thread is defined by the lambda expression at (3). The task executes an infinite `while` loop at (4) that illustrates how a thread can discover it has received an interrupt:

- An `if` statement at (5), which is used to determine if the thread has been interrupted. This corresponds to the first scenario in the output.
  If any interrupt is sent to the current thread by calling its `interrupt()` method while it is running, the interrupt status of the thread will be set. Calling the `isInterrupted()` method on the current thread in the `if` statement will then return `true`.
- A `try-catch` block at (6) in which the thread sleeps, and catches any `InterruptedException` that is thrown in response to receiving an interrupt while it was asleep. This corresponds to the second scenario in the output.
  If the `interrupt()` method was called on the thread while it was sleeping, it will result in the interrupt status being cleared and an `InterruptedException` being thrown in the `try` block when the thread runs again. Appropriate action can be taken to mitigate the interrupt in the `catch` block at (7) in which this exception is caught.

Example 22.5 *Interrupt Handling*

Click here to view code image

```
public class InterruptHandling {
  public static void main(String[] args) throws InterruptedException {
    Thread worker = new Thread(task, "worker");                    // (1)
    worker.start();
    while (true) {                                                 // (2)
      if ((int)(Math.random()*100) % 2 == 0) {
        worker.interrupt();
        break;
      }
      Thread.sleep(2);
    }
  }
```

```
  private static Runnable task = () -> {                          // (3)
    Thread ct = Thread.currentThread();
    String threadName = ct.getName();
    while (true) {                                                // (4)
      System.out.println(threadName + " performing task");

      if (ct.isInterrupted()) {                                   // (5)
        System.out.println(threadName + ": interrupted flag is "
                           + ct.isInterrupted());
        System.out.println(threadName + " terminating");
        return;
      }

      try {                                                        // (6)
        Thread.sleep(2);
      } catch (InterruptedException e) {                           // (7)
        System.out.println(threadName + " caught " + e);
        System.out.println(threadName + ": interrupted flag is "
                           + ct.isInterrupted());
        System.out.println(threadName + " terminating");
        return;
      }
    }
  };
}
```

Probable output from the program—first scenario:

Click here to view code image

```
worker performing task
worker: interrupted flag is true
worker terminating
```

Probable output from the program—second scenario:

Click here to view code image

```
worker performing task
worker performing task
worker caught java.lang.InterruptedException: sleep interrupted
worker: interrupted flag is false
worker terminating
```

Sleeping and Waking Up

Transitions by a thread that is sleeping and waking up are illustrated in Figure 22.8.

A call to the static method `sleep()` in the `Thread` class will cause the *currently running* thread to temporarily pause its execution and transit to the `TIMED_WAITING` state. The `getState()` method on this thread will return the value `Thread.State.TIMED_WAITING`.

The thread is *guaranteed* to sleep for *at least* the time specified in its argument, before transitioning to the *READY* substate where it takes its turn to run again. It might sleep a little longer than the specified time, but it will not sleep indefinitely.

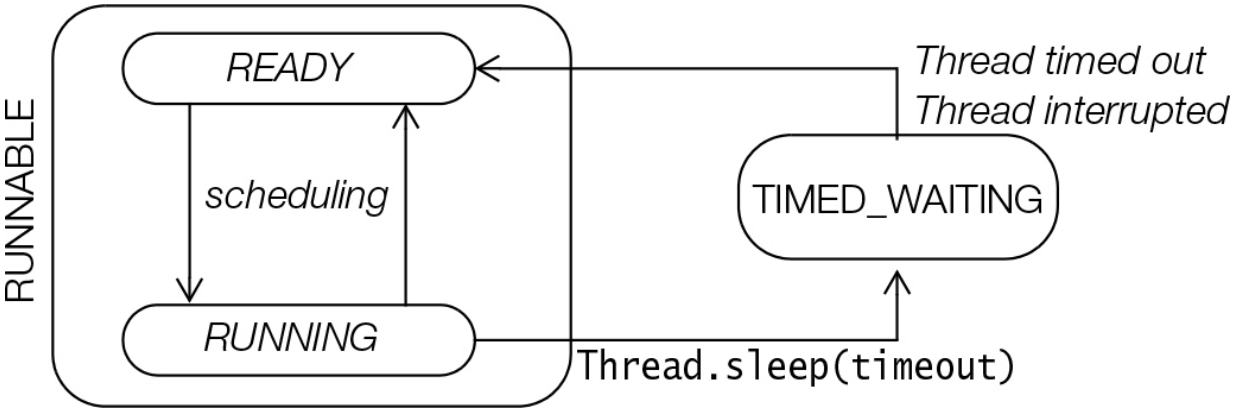

Figure 22.8 *Sleeping and Waking Up*

If a thread is interrupted while sleeping, it will throw a checked `InterruptedException` when it awakes and gets to execute. The idiom is to call the `sleep()` method in a `try-catch` block that handles the `InterruptedException`. The `sleep()` method does *not* release any locks that the thread might have.

There are two overloaded versions of the `sleep()` method in the `Thread` class, allowing time to be specified in milliseconds, and additionally in nanoseconds.

Usage of the `sleep()` method is illustrated in **Examples 22.1**, 22.2, and 22.4.

Thread Coordination

Before proceeding further with thread coordination using the `wait()` and `notify()` / `notifyAll()` methods, it must be emphasized that these methods provide *low-level* thread coordination, which is error-prone and difficult to get right, and should be avoided in favor of the *high-level* concurrency support provided by the `java.util.concurrent` package (**Chapter 23, p. 1419**).

Waiting and notifying provide a means of coordination between threads that *synchronize on the same object*—also called *data access synchronization.* The threads execute `wait()` and `notify()` (or `notifyAll()`) methods *on the shared object* for this purpose. These `final` methods are defined in the `Object` class, and therefore are inherited by all objects. These methods can only be executed on an object whose lock the thread holds (i.e., in `synchronized` code); otherwise, the call will result in an `IllegalMonitorStateException`.

Click here to view code image

```
final void wait(long timeout) throws InterruptedException
final void wait(long timeout, int nanos) throws InterruptedException
final void wait() throws InterruptedException
```

A thread invokes the `wait()` method on the object whose lock it holds. The thread is added to the *wait set* of the current object.

```
final void notify()
final void notifyAll()
```

A thread invokes a notification method on the current object whose lock it holds to notify thread(s) that are in the wait set of the object.

Waiting

Coordination between threads is facilitated by waiting and notifying, as illustrated by **Figures 22.9** and **22.10**. A thread usually calls the `wait()` method on the object whose lock it holds because a condition for its continued execution was not met. The thread leaves the *RUNNING* substate and transits to the `WAITING` or `TIMED_WAITING` state, depending on whether a timeout was specified in the call. The thread releases ownership of the object lock.

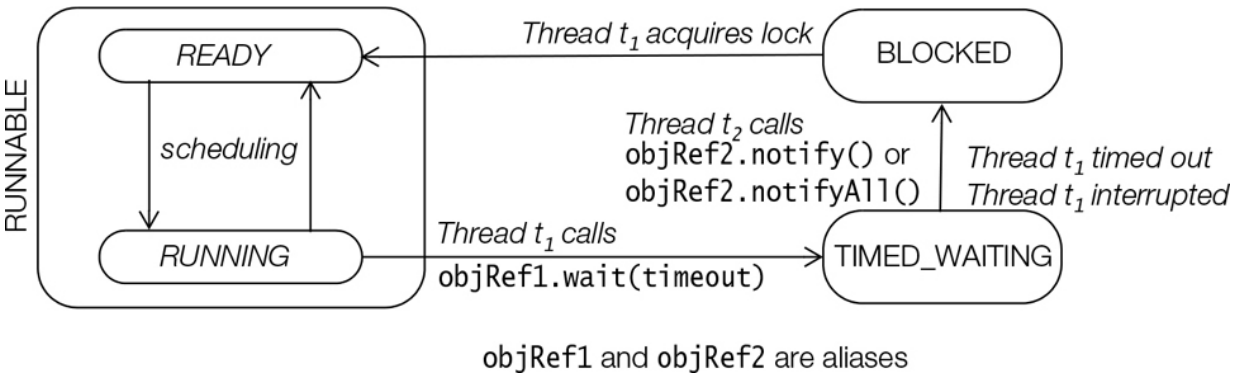

Figure 22.9 *Timed Waiting and Notifying*

Transitioning to a waiting state and releasing the object lock are completed as one *atomic* (non-interruptible) operation. The releasing of the lock of the shared object by the thread allows other threads to run and execute synchronized code on the same object, after acquiring its lock.

Note that the waiting thread releases only the lock of the object on which the `wait()` method was invoked. It does not release any other object locks that it might hold, which will remain locked while the thread is waiting.

Each object has a *wait set* containing threads waiting for notification. Threads in a waiting state are grouped according to the object whose `wait()` method they invoked.

Figure 22.10 shows a thread t_1 that first acquires a lock on the shared object, and afterward invokes the `wait()` method on the shared object. This releases the object lock, the thread t_1 is added to the wait set of the object, and the thread waits to be notified in the `WAITING` state. While the thread t_1 is waiting, another thread t_2 can acquire the lock on the shared object for its own purpose.

Depending on whether a timeout was specified or not in the call to the `wait()` method, a call to the `getState()` method on thread t_1 while it is in a waiting state will return the value `Thread.State.TIMED_WAITING` or `Thread.State.WAITING`, respectively.

A thread in a waiting state can be awakened by the occurrence of any one of these events:

- Another thread invokes the `notify()` method on the object of the waiting thread, and the waiting thread is selected as the thread to be awakened.
- The waiting thread times out if it was a timed wait.
- Another thread interrupts the waiting thread.
- A spurious wakeup occurs.

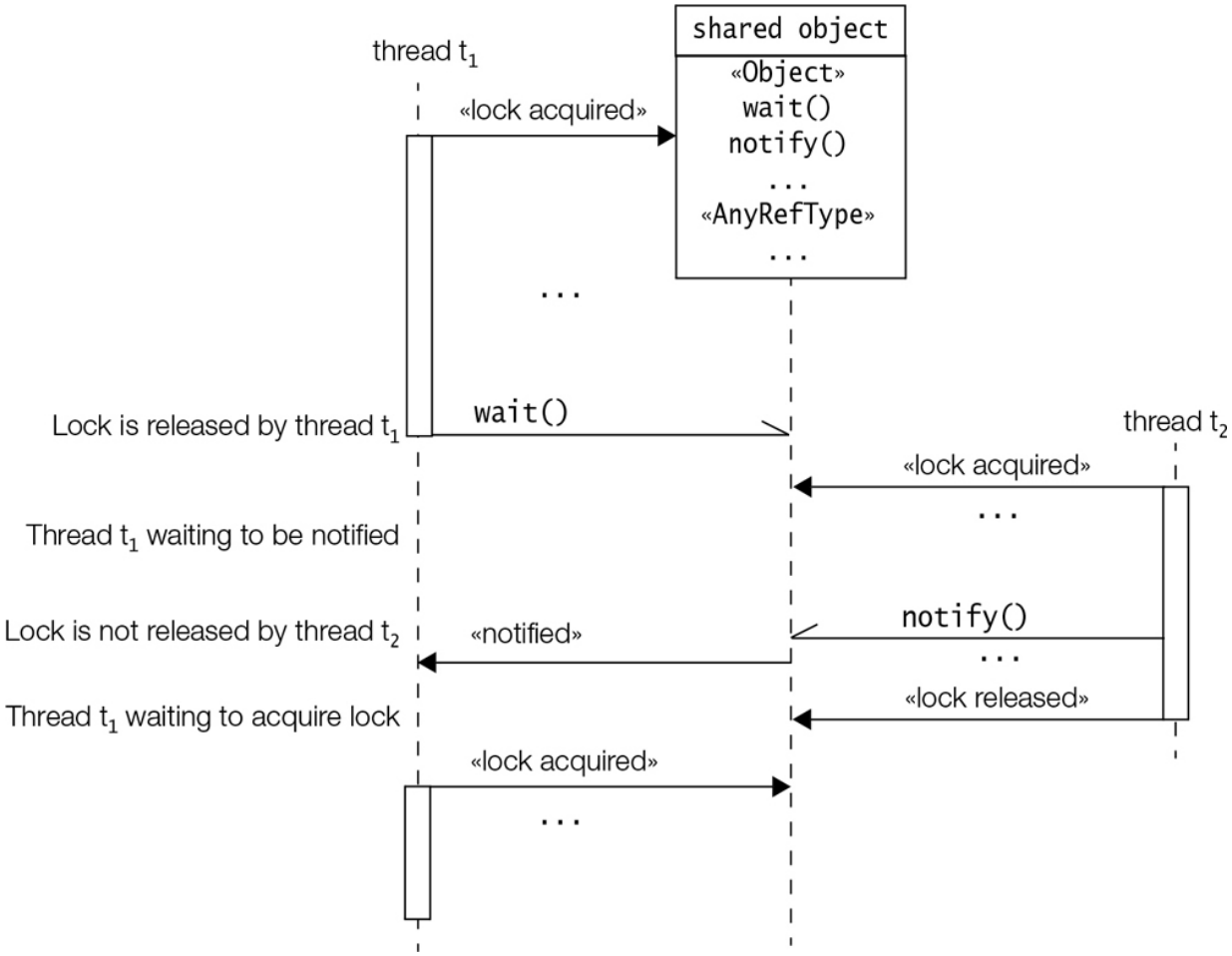

Figure 22.10 *Thread Coordination*

Notified

Invoking the `notify()` method on an object wakes up a single thread that is waiting for the lock of this object. The selection of a thread to awaken is dependent on the thread policies implemented by the JVM. On being *notified*, a waiting thread first transits to the `BLOCKED` state to acquire the lock on the object, and not directly to the *READY* substate of the

`RUNNABLE` state. The thread is also removed from the wait set of the object. Note that the object lock is not released when the notifying thread invokes the `notify()` method. The notifying thread releases the lock at its own discretion, and the awakened thread will not be able to run until the notifying thread releases the object lock.

When the notified thread obtains the object lock, it is enabled for execution, waiting in the *READY* substate for its turn to execute again. Finally, when it does execute, the call to the `wait()` method returns and the thread can continue with its execution.

From **Figure 22.10** we see that thread t_2 does not release the object lock when it invokes the `notify()` method. Thread t_1 is forced to wait in the `BLOCKED` state for lock acquisition. It is shown no privileges, and must compete with any other threads blocked for lock acquisition.

A call to the `notify()` method has no effect if there are no threads in the wait set of the object.

In contrast to the `notify()` method, the `notifyAll()` method wakes up *all* threads in the wait set of the shared object. They will all transit to the `BLOCKED` state, and contend for the object lock as explained earlier.

It should be stressed that a program should not make any assumptions about the order in which threads awaken in response to the `notify()` or `notifyAll()` method, or transit to the `BLOCKED` state for lock acquisition.

Timed-out

The `wait()` call can specify the time the thread should wait before being timed out, if it was not awakened by being notified or interrupted. The timed-out thread transits to the `BLOCKED` state and competes to acquire the lock as explained above. Note that the awakened thread has no way of knowing whether it was timed out or awakened by one of the notification methods.

If no timeout is specified in the call to the `wait()` method, the thread waits indefinitely in the `WAITING` state until it is notified, interrupted, or awakened spuriously.

Interrupted

This means that another thread invoked the `interrupt()` method on the waiting thread. The awakened thread is enabled as previously explained, but the return from the `wait()` call will result in an `InterruptedException` if and when the awakened thread finally gets a chance to run. The code invoking the `wait()` method must be prepared to handle this checked exception (**p. 1393**).

Spurious Wakeup

In very rare cases, a thread in a waiting state is awakened by what is called a *spurious wakeup*, which is not due to familiar events like being notified, timed out, or interrupted, but due to deep-level system signals warranting a wakeup call to the waiting threads. The condition on which the thread was waiting might not have been fulfilled when the thread wakes up from a spurious wakeup, and should therefore be rechecked when the thread gets to run. This is done by testing the condition in a loop that contains the `wait()` call, as the thread can go right back to waiting if the condition is not satisfied.

Click here to view code image

```
while (conditionNotSatisfied()) {
  ...
  wait();
  ...
}
// Proceed ...
```

Using Wait and Notify

The idiom is to call the `wait()` method in a `try-catch` block inside a loop, guarded by a condition that the thread expects to be set by notification.

In Example 22.6, the `main()` method of the class `WaitNotifyScenario` creates and starts four daemon threads that are manipulating the same `MessageDisplay` object. Two of them are continuously trying to set a message in the message display, while the other two are continuously trying to print whatever message is in the display.

Since the threads manipulate the same `MessageDisplay` object, and the `setMessage()` and `displayMessage()` methods in the class `MessageDisplay` are `synchronized`, it means that the threads synchronize on the same `MessageDisplay` object. In other words, the mutual exclusion of these operations is guaranteed on the same `MessageDisplay` object.

Example 22.6 *Waiting and Notifying*

Click here to view code image

```
class MessageDisplay {
  private String message;

  public synchronized void displayMessage() {
    String threadName = Thread.currentThread().getName();
    while (this.message == null) {                          // No message?
      try {
        wait();                                             // (1)
      } catch (InterruptedException e) {
        e.printStackTrace();
```

```
      }
    }
    System.out.println(threadName + ": " + this.message);    // Display message.
    this.message = null;                                     // Remove message.
    notifyAll();                                             // (2)
  }

  public synchronized void setMessage(String message) {
    String threadName = Thread.currentThread().getName();
    while (this.message != null) {                           // Message present?
      try {
        wait();                                              // (3)
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    this.message = message;                                  // Set new message.
    System.out.println(threadName + ": Message set is " + this.message);
    notifyAll();                                             // (4)
  }
}
```

```
public class WaitNotifyScenario {
  public static void main(String[] args) throws InterruptedException {
    MessageDisplay md = new MessageDisplay();
    Thread t1 = new Thread(() -> { while (true) md.setMessage("Hi!"); }, "t1");
    t1.setDaemon(true);
    t1.start();
    Thread t2 = new Thread(() -> { while (true) md.setMessage("Howdy!"); }, "t2");
    t2.setDaemon(true);
    t2.start();
    Thread t3 = new Thread(() -> { while (true) md.displayMessage(); }, "t3");
    t3.setDaemon(true);
    t3.start();
    Thread t4 = new Thread(() -> { while (true) md.displayMessage(); }, "t4");
    t4.setDaemon(true);
    t4.start();
    Thread.sleep(5);
    System.out.println("Exit from main.");
  }
}
```

Probable output from the program (*edited to fit on the page*):

```
t0: t1: Message set is Hi!
t3: Hi!
...
t2: Message set is Howdy!
t3: Howdy!
t1: Message set is Hi!
```

```
t3: Hi!
t2: Message set is Howdy!
t4: Howdy!
...
Exit from main.
t1: Message set is Hi!
t3: Hi!
t2: Message set is Howdy!
t4: Howdy!
t1: Message set is Hi!
```

Example 22.6 illustrates how threads waiting as a result of calling the `wait()` method on an object are notified by another thread calling the `notifyAll()` method on the same object, in order for a waiting thread to start running again.

One usage of the `wait()` call is shown in **Example 22.6** at (1) in the `synchronized displayMessage()` method. When a thread executing this method on the `Message-Display` object finds that there is no message (`this.message == null`), it invokes the `wait()` method in order to wait for a thread to set a message.

Another use of the `wait()` call is shown at (3) in the `synchronized setMessage()` method. When a thread executing this method on the `MessageDisplay` object finds that there is already a message (`this.message != null`), it invokes the `wait()` method to wait for a thread to display and remove the message.

When a thread executing the `synchronized` method `setMessage()` on the `MessageDisplay` object successfully sets a message, it calls the `notifyAll()` method at (4). The wait set of the `MessageDisplay` object may contain any waiting threads that have earlier called the `wait()` method at either (1) or (3) on this `MessageDisplay` object. A single thread from the wait set is enabled for running. If this thread was executing a `displayMessage()` operation, it now has a chance of being successful because a message is available for display at the moment. If this thread was executing a `setMessage()` operation, it can try again to see if the previous message has been removed so that a new message can be set.

When a thread executing the `synchronized` method `displayMessage()` on the `MessageDisplay` object successfully prints and removes the message, it calls the `notifyAll()` method at (2). Again assuming that the wait set of the `MessageDisplay` object is not empty, one thread from the set is arbitrarily chosen and enabled. If the notified thread was executing a `setMessage()` operation, it now has a chance of succeeding because there is no message in the `MessageDisplay` object at the moment.

Note that the waiting condition at (1) for the `displayMessage()` operation is executed in a loop. A waiting thread that has been notified is not guaranteed to run right away. Before it gets to run, another thread may synchronize on the `MessageDisplay` object and remove the message. If the notified thread was waiting to display the message, it would now incorrectly try to display a nonexisting message because the condition was not tested after notification. The loop ensures that the condition is always tested after notification, sending the thread

back to the waiting state if the condition is not met. To avert the analogous danger of setting a message if one already exits in the `MessageDisplay` object, the waiting condition at (3) for the `setMessage()` operation is also executed in a loop.

The output from **Example 22.6** shows the behavior of the threads that set and print the message in the shared `MessageDisplay` object, respectively. The two operations of setting the message and displaying/removing the message are performed in lockstep.

The four threads created are daemon threads. They will be terminated if they have not completed when the main thread dies, thereby stopping the execution of the program.

Joining

A thread can invoke the overloaded method `join()` (from the `Thread` class) on another thread in order to wait for the other thread to complete its execution before continuing—that is, the first thread waits for the second thread to *join it after completion*.

A running thread t_1 invokes the method `join()` on a thread t_2. The `join()` call has no effect if thread t_2 has already completed. If thread t_2 is still alive, thread t_1 transits to one of the two waiting states. Depending on whether a timeout was specified or not in the call to the `join()` method, a call to the `getState()` method on thread t_1 while it is waiting for join completion will return the value `Thread.State.TIMED-_WAITING` or `Thread.State.WAITING`, respectively.

Thread t_1 waits in a waiting state until one of these events occurs (**Figure 22.11**):

- Thread t_2 completes.
  In this case, thread t_1 moves to the *READY* substate, and when it gets to run, it will continue normally after the call to the `join()` method.
- Thread t_1 is timed out.
  The time specified in the argument of the `join()` method call has elapsed without thread t_2 completing. In this case as well, thread t_1 transits from the `TIMED_WAITING` state to the *READY* substate. When it gets to run, it will continue normally after the call to the `join()` method.
  If the no-argument `join()` method was called, there is no timeout for thread t_1 in the `WAITING` state. It waits indefinitely in this state for one of the other events to occur.
- Thread t_1 is interrupted.
  Some thread interrupted thread t_1 while it was waiting for join completion. Thread t_1 transits to the *READY* substate, but when it gets to execute, it will now receive an `InterruptedException`.

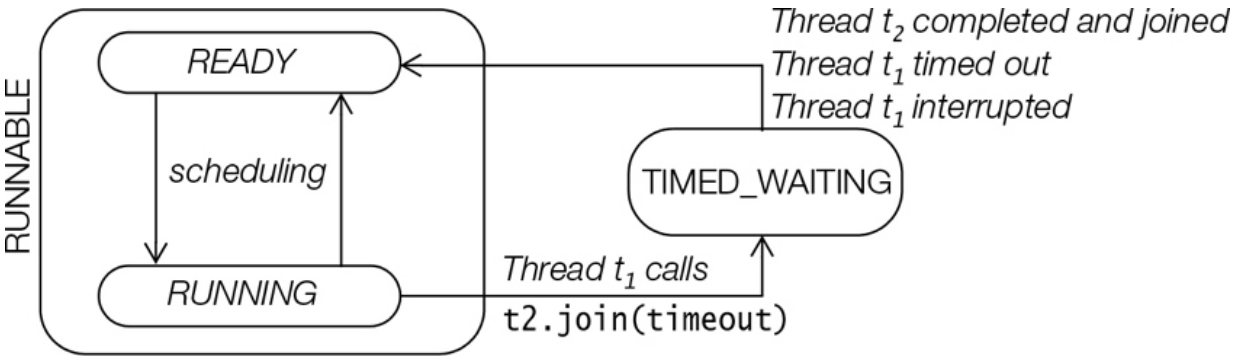

Figure 22.11 *Timed Joining of Threads*

Example 22.7 illustrates joining of threads. The `AnotherClient` class below uses the `Counter` class, which extends the `Thread` class, from Example 22.2. It creates two threads that are enabled for execution. The main thread invokes the `join()` method on the `Counter A` thread. If the `Counter A` thread has not already completed, the main thread transits to the `WAITING` state, as no timeout is specified. When the `Counter A` thread completes, the main thread will be enabled for running. Once the main thread is running, it continues with execution after (5). A parent thread can call the `isAlive()` method to find out whether its child threads are alive, before terminating itself. The call to the `isAlive()` method on the `Counter A` thread at (6) correctly reports that the `Counter A` thread is not alive. A similar scenario transpires between the main thread and the `Counter B` thread. At most, the main thread passes through the `WAITING` state twice.

Example 22.7 *Joining of Threads*

Click here to view code image

```
class Counter extends Thread { /* See Example 22.2, p. 1376. */ }
```

Click here to view code image

```
public class AnotherClient {
  public static void main(String[] args) {

    // Create two Counter threads, set their names, and start them:    // (4)
    Counter counterA = new Counter();
    Counter counterB = new Counter();
    counterA.setName("counterA");
    counterB.setName("counterB");
    counterA.start();
    counterB.start();

    try {
      System.out.println("Wait for the child threads to finish.");
      counterA.join();                                                  // (5)
      if (!counterA.isAlive()) {                                        // (6)
        System.out.println("Counter A not alive.");
      }
      counterB.join();                                                  // (7)
```

```
      if (!counterB.isAlive()) {
        System.out.println("Counter B not alive.");
      }
    } catch (InterruptedException ie) {
      System.out.println("main thread interrupted.");
    }
    System.out.println("Exiting from main thread.");
  }
}
```

Probable output from the program:

Click here to view code image

```
Wait for the child threads to finish.
counterB: 0
counterA: 0
counterA: 1
counterB: 1
counterA: 2
counterB: 2
counterA: 3
counterB: 3
counterA: 4
counterB: 4
Exiting counterB
Exiting counterA
Counter A not alive.
Counter B not alive.
Exiting from main thread.
```

Blocking for I/O

A running thread, on executing a *blocking operation* requiring a resource (like a call to an I/O method), will transit to the `BLOCKED` state (**Figure 22.12**). The `getState()` method on this thread will return the value `Thread.State.BLOCKED`. When the thread can complete the blocking operation, it proceeds to the *READY* substate. An example is a thread reading from the standard input terminal, that blocks until input is provided:

```
int input = System.in.read();
```

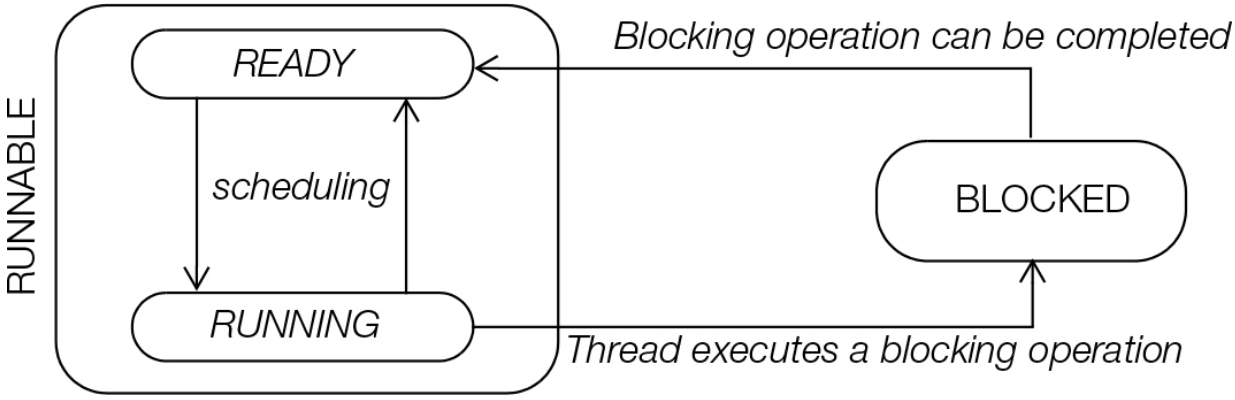

Figure 22.12 *Blocked for I/O*

Thread Termination

A thread can transit to the `TERMINATED` state from the `RUNNABLE` state (**Figure 22.13**). The thread terminates when it finishes executing its `run()` method, either by returning normally or by throwing an exception. Once in this state, the thread cannot be resurrected. There is no way the thread can be enabled for running again, not even by calling the `start()` method again on the thread object. The `getState()` method on this thread will return the value `Thread.State.TERMINATED`.

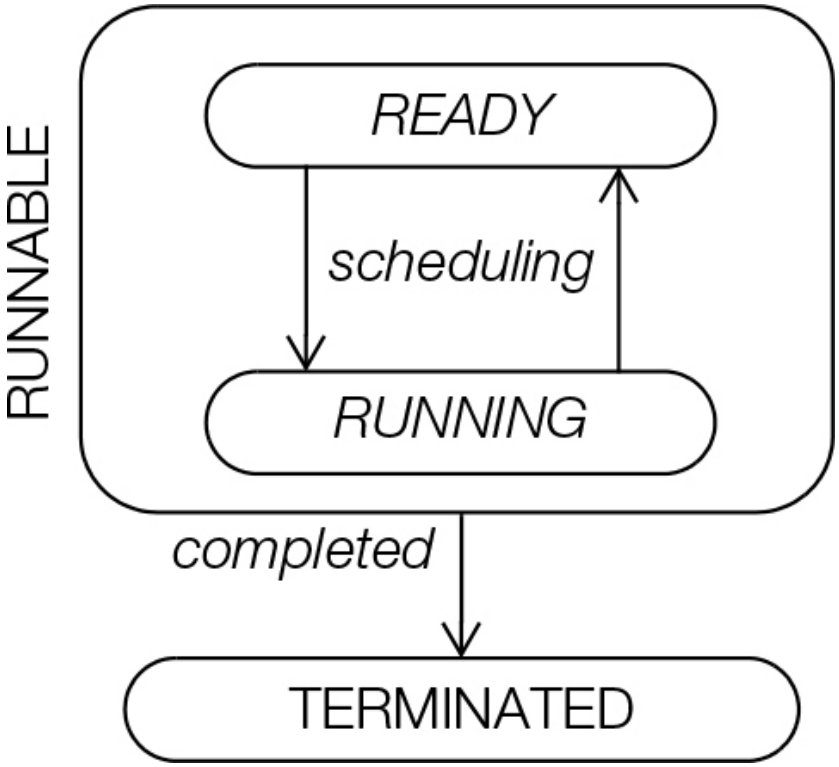

Figure 22.13 *Thread Termination*

The *main thread* is special in that it is started and terminated when the `main()` method starts and stops execution, respectively.

It might be tempting to call the `System.exit()` method to terminate a thread, but this can be rather drastic. A call to this method terminates the *process* in which the JVM is running, and as a consequence, terminating all threads that might still be alive in the JVM.

Controlling a Thread

Example 22.8 illustrates a typical scenario where a thread can be controlled by one or more threads. Work is performed by a loop body, which the thread executes continuously. It should be possible for other threads to start and stop the *worker* thread. This functionality is implemented by the class `Worker` at (1), which has a private field `theThread` declared at (2) to keep track of the `Thread` object executing its `run()` method at (5).

The `kickStart()` method at (3) in the class `Worker` creates and starts a thread if the field `theThread` is not already denoting a running thread—that is, if the field has the `null` value. The `terminate()` method at (4) sets the field `theThread` to `null` to signal that the `run()` method can terminate, thus resulting in the thread being terminated. Note that this does not affect any `Thread` object that might have been referenced by the field `theThread`. The runtime system maintains any such `Thread` object; therefore, changing one of its references does not affect the `Thread` object.

The `run()` method at (5) has a loop whose execution is controlled by a special condition. The condition tests to see whether the `Thread` object referenced by the reference `theThread` and the `Thread` object currently running are one and the same. This is bound to be the case if the reference `theThread` has the same reference value that it was assigned when the thread was created and started in the `kickStart()` method. The condition will then be `true`, and the body of the loop will execute. However, if the value in the reference `theThread` has changed, the condition will be `false`. In that case, the loop will not execute, the `run()` method will complete, and the thread will terminate. This idiom is generally recommended to terminate a thread.

A client can control the thread implemented by the class `Worker`, using the `kick-Start()` and the `terminate()` methods. The client is able to terminate the running thread at the start of the next iteration of the loop body by calling the `terminate()` method that changes the value of the `theThread` reference to `null`.

In **Example 22.8**, a `Worker` object is first created at (8) and a thread started on this `Worker` object at (9). The main thread invokes the `sleep()` method at (10) to temporarily cease its execution for 2 milliseconds, giving the thread of the `Worker` object a chance to run. The main thread, when it is executing again, terminates the thread of the `Worker` object at (11), as explained earlier. This simple scenario can be generalized where several threads, sharing a single `Worker` object, could be starting and stopping the thread of the `Worker` object. However, this generalization also requires that the field `theThread` is declared `volatile` in order to avoid memory consistency errors (p. 1414).

Example 22.8 *Thread Termination*

Click here to view code image

```
public class Worker implements Runnable {                          // (1)
  private Thread theThread;                                         // (2)
```

```
  public void kickStart() {                                    // (3)
    if (theThread == null) {
      theThread = new Thread(this);
      theThread.start();
    }
  }

  public void terminate() {                                    // (4)
    theThread = null;
  }

  @Override
  public void run() {                                          // (5)
    while (theThread == Thread.currentThread()) {              // (6)
      System.out.println("Going around in loops.");
    }
  }
}
```

Click here to view code image

```
public class Controller {
  public static void main(String[] args) {                     // (7)
    Worker worker = new Worker();                              // (8)
    System.out.println("Start the worker.");
    worker.kickStart();                                        // (9)
    try {
      Thread.sleep(2);                                         // (10)
    } catch(InterruptedException ie) {
      ie.printStackTrace();
    }
    System.out.println("Stop the worker.");
    worker.terminate();                                        // (11)
  }
}
```

Probable output from the program:

```
Start the worker.
Going around in loops.
Going around in loops.
...
Going around in loops.
Going around in loops.
Stop the worker.
Going around in loops.
```

Deprecated Thread Methods

There are a few operations defined by the `Thread` class that are not recommended for use and are now marked as `@Deprecated`. These include the following methods of the `Thread` class:

```
final void resume()
final void stop()
final void suspend()
```

Their initial design intention was to control the thread lifecycle. However, Java concurrency design has changed since these operations were introduced. The new concurrency API allows concurrent programming at a higher level than controlling the threads. The legacy approach to controlling threads was to force a thread to become suspended, to resume, or to stop. Deprecated concurrency methods are likely to result in memory corruption. The code in this chapter does not use any of the deprecated methods.

22.5 Thread Issues

In this section, we briefly outline some problems that occur often in multithreaded applications. Although some solutions are mentioned, there is no silver bullet that will solve all such problems. In the worst-case scenario, the application may need to be redesigned all over again.

Liveness and Fairness

Multithreaded applications strive for *liveness* and *fairness*.

The *fairness property* of a multithreaded application refers to threads in the application getting a fair chance to run—so that all threads in the application can make progress in their work, and no thread monopolizes the CPU at the expense of others.

The *liveness property* of a multithreaded application refers to the ability of the threads to execute in a *timely manner*—meaning performing as expected and making continuous progress in their work.

The rest of this section describes three liveness issues common in multithreaded applications: *deadlock*, *livelock*, and *starvation*.

Deadlock

A *deadlock* is a situation where threads are holding locks on objects that other threads need—a thread is waiting for an object lock that another thread holds, and this second thread is

waiting for an object lock that the first thread holds. Since each thread is waiting for the other thread to release a lock, they both remain waiting forever in a waiting state and never make any progress. The threads are said to be *deadlocked*.

A deadlock is depicted in **Figure 22.14**. Thread t_1 has a lock on object o_1, but cannot acquire the lock on object o_2. Thread t_2 has a lock on object o_2, but cannot acquire the lock on object o_1. They can only proceed if one of them releases a lock the other one wants, which is never going to happen.

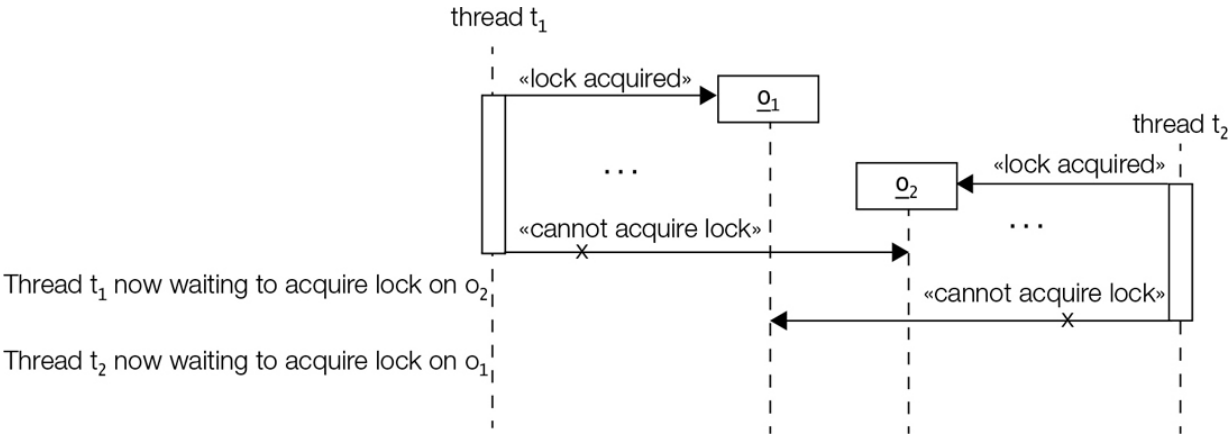

Figure 22.14 *Deadlock*

The situation in **Figure 22.14** is implemented in **Example 22.9**. Thread `t1` at (3) tries to synchronize at (4) and (5), first on string `o1` at (1), then on string `o2` at (2), respectively. The thread `t2` at (6) does the opposite. It synchronizes at (7) and (8), first on string `o2` and then on string `o1`, respectively. A deadlock can occur as explained previously, and the print statement at (11) will never be executed.

However, the potential of a deadlock in the situation in **Example 22.9** is easy to fix. If the two threads acquire the locks on the objects in the same order, then mutual lock dependency is avoided and a deadlock can never occur. This means having the same locking order at (4) and (5) as at (7) and (8). In general, the cause of a deadlock is not always easy to discover, let alone easy to fix.

Example 22.9 *Deadlock*

Click here to view code image

```
public class DeadLockDanger {
  public static void main(String[] args) {
    String o1 = "o1 " ;                                          // (1)
    String o2 = "o2 ";                                           // (2)

    Thread t1 = (new Thread("t1") {                              // (3)
      @Override
      public void run() {
        String threadName = Thread.currentThread().getName();
        while (true) {
          synchronized(o1) {                                     // (4)
```

```
            System.out.println(threadName + " acquired " + o1);
            synchronized(o2) {                                  // (5)
              System.out.println(threadName + ": " + o1 + o2);
      }}}}});

    Thread t2 = (new Thread("t2") {                             // (6)
      @Override
      public void run() {
        String threadName = Thread.currentThread().getName();
        while (true) {
          synchronized(o2) {                                    // (7)
            System.out.println(threadName + " acquired " + o2);
            synchronized(o1) {                                  // (8)
              System.out.println(threadName + ": " + o2 + o1);
      }}}}});

    t1.start();                                                 // (9)
    t2.start();                                                 // (10)
    System.out.println("Exiting main.");                        // (11)
  }
}
```

It is possible that the program in **Example 22.9** might run without a deadlock, depending on how the locks on the two `String` objects are acquired by each thread. However, after the following output from the program, a deadlock occurs and the program never terminates:

```
t1 acquired o1
t2 acquired o2
```

Livelock

Two threads may result in a livelock, if the response by a thread to another thread's action is always to undo or revert the consequences of the action. The threads are responding to each other, but are blocked in making any progress. The combined actions of the threads make it impossible for them to complete their tasks, making it impossible for the threads to terminate. This situation is different from a deadlock, where the threads are blocked, waiting for locked shared resources to become available.

An example of a livelock is when two guests (i.e., two threads) are trying to enter through a door, but there is only one door key and only one guest at a time can use it. Each guest is too polite, and insists on giving the key to the other if the other wants to enter. In other words, if one guest hands the key to the other, the other hands it right back. This situation will continue indefinitely, with neither guest thereby entering through the door. We have a livelock because of handing the key back and forth.

Example 22.10 provides another example of a livelock. Two threads are created at (11) and (12). In the first one, a customer makes a payment to a seller, and in the second one, the seller

ships an item to the customer.

The `Customer` class at (1) has a flag (`paymentMade`) to mark that payment has been made. In the `makePaymentTo()` method, the customer continuously checks whether the seller has sent the shipment. If the seller has not, the customer waits before checking again. Once the seller has shipped the item, the customer marks at (4) that the payment has now been made.

On the other hand, the `Seller` class at (6) has a flag (`itemShipped`) to record that shipment has been sent. In the `shipTo()` method, the seller continuously checks whether the customer has paid. If the customer has not, the seller waits before checking again. Once the customer has paid, the seller marks at (5) that the shipment has been sent.

Output from running the program shows that the customer is waiting for shipment from the seller, and the seller is waiting for payment from the customer. Neither thread is able to make progress because the seller cannot send the shipment before the customer has paid, and the customer cannot pay before the shipment has been sent. Each is expecting a confirmation from the other, which never arrives. They need to renegotiate their contract, and one of them will have to trust the other.

Example 22.10 *Livelock*

Click here to view code image

```
public class Customer {                                                    // (1)
  private boolean paymentMade = false;                                     // (2)

  public void makePaymentTo(Seller seller) {                               // (3)
    while (!seller.hasShipped()) {
      System.out.println("Customer: waiting for shipment from seller");
      try {
        Thread.sleep(1000);
      } catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
    System.out.println("Customer: payment made");
    this.paymentMade = true;                                               // (4)
  }

  public boolean hasPaid() {
    return this.paymentMade;
  }
}
```

Click here to view code image

```
public class Seller {                                                      // (5)
  private boolean itemShipped = false;                                     // (6)
```

```
  public void shipTo(Customer customer) {                                  // (7)
    while (!customer.hasPaid()) {
      System.out.println("Seller: waiting for payment from customer");
      try {
        Thread.sleep(500);
      } catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
    System.out.println("Seller: item shipped");
    this.itemShipped = true;                                               // (8)
  }

  public boolean hasShipped() {
    return this.itemShipped;
  }
}
```

Click here to view code image

```
public class LivelockShipment {
  public static void main(String[] args) {
    Customer customer = new Customer();
    Seller seller = new Seller();

    new Thread(() -> customer.makePaymentTo(seller)).start();              // (9)
    new Thread(() -> seller.shipTo(customer)).start();                     // (10)
  }
}
```

Output from the program:

Click here to view code image

```
...
Seller: waiting for payment from customer
Customer: waiting for shipment from seller
Seller: waiting for payment from customer
Seller: waiting for payment from customer
Customer: waiting for shipment from seller
Seller: waiting for payment from customer
...
```

Starvation

A thread can *starve* because it is unsuccessfully waiting for its turn to be able to proceed with its execution. In addition to deadlock and livelock that result in starvation, some other situations where starvation can occur are the following:

- *Unfair priority scheduling*: Higher-priority threads monopolize the CPU, and low-priority threads starve in the *READY* substate.
- *Indefinite waiting for join to complete:* A thread that executed a no-timeout `join()` call can starve in the `WAITING` state if the thread it is waiting for never completes.
- *Indefinite waiting in the entry set of the object lock:* A thread can starve in the `BLOCKED` state if other threads get chosen before it to acquire the object lock to enter synchronized code.
- *Indefinite waiting in the wait set of an object*: A thread that executed a no-timeout `wait()` call can starve in the `WAITING` state if it never gets notified. It can also starve in this state if other threads get chosen on notification before it does, and it never gets to compete for object lock acquisition.

Example 22.11 provides another example of starvation. The `Hole` class at (1) provides the `synchronized` method `dig()` to dig a hole. The class `Diggers` creates a `Hole` object and starts five threads to dig this hole. The output shows that the thread that first acquires the lock to execute the `synchronized dig()` method in the `Hole` object monopolizes the CPU, and the other threads do not get a chance to run—they wait and starve.

The starvation scenario in **Example 22.11** can be remedied by making a thread wait between digging—implemented by the `try-catch` block at (3) with the call to the *timed* `wait()` method. The call to the `wait()` method will result in the current thread transiting to the `TIMED_WAITING` state and releasing the lock on the `Hole` object, thus making it possible for other waiting threads to run. On the other hand, a call to the `sleep()` method does not release the lock and will still result in thread starvation.

Example 22.11 *Starvation*

Click here to view code image

```
public class Hole {                                                    // (1)
  public synchronized void dig() {
    String threadName = Thread.currentThread().getName();
    while (true) {                                                     // (2)
      System.out.println(threadName + " digging the hole.");
//    try {                                                            // (3)
//    wait(1);
//    } catch (InterruptedException ex) {
//      ex.printStackTrace();
//    }
    }
  }
}
```

Click here to view code image

```
public class Diggers {
  public static void main(String[] args) {
```

```
    Hole hole = new Hole();                                    // (4)
    for (int i = 0; i < 5; i++) {                              // (5)
      new Thread(() -> hole.dig()).start();
    }
  }
}
```

Probable output from the program:

```
...
Thread-0 digging the hole.
Thread-0 digging the hole.
Thread-0 digging the hole.
Thread-0 digging the hole.
Thread-0 digging the hole.
Thread-0 digging the hole.
...
```

Memory Consistency Errors

In a multithreaded application, there is a potential danger that changes made by one thread to shared data might not be *visible* to the other threads, resulting in inconsistent views of the shared data. When this happens, it is known as a *memory consistency error*.

Causes of memory consistency errors are too complex to go into here. The following simple example gives an idea of how they can manifest. Two threads have access to a shared counter.

Click here to view code image

```
int counter = 0;

// Thread A                                    // Thread B
...                                            ...
counter = 1;                                   ...
System.out.println("Thread A:" + counter);     System.out.println("Thread B:" + counter);
...                                            ...
```

Thread `A` set the counter value to `1` and prints it value. We can safely assume that the value printed would be `1`—that is, the value in the counter set by the preceding assignment statement. However, if the print statement is executed by thread `B`, there is no guarantee that the change in the counter value in thread `A` will be visible to thread `B`—it might print `0` or `1`—unless it can be established that a change of value in thread `A` occurred before being printed in thread `B`.

Happens-Before Relationship

The *happens-before relationship* helps to combat memory consistency errors. It is important in guaranteeing that *one action is ordered before another* in a multithreaded runtime environment. More importantly, the first action is *visible* to the second one, meaning that the *results* of the first action are evident to the second action. The relationship is *transitive*: If action `A` happens-before action `B`, and action `B` happens-before action `C`, then action `A` happens-before action `C`.

Little is guaranteed in a multithreaded environment, and therefore, the happens-before relationship is important in ascertaining certain properties about multi-threaded applications. Here are some important rules pertaining to multithreaded applications in Java:

- *Object lock rule*: An unlock action on a monitor happens-before every subsequent lock action on that monitor.
- *Volatile field rule*: A write to a `volatile` field happens-before every subsequent read of that field (§23.4, p. 1453).
- *Thread start rule*: A call to the `Thread.start()` method on a thread happens-before any actions in the started thread.
- *Thread termination rule*: All actions in a thread happen-before any other thread can detect that the thread has terminated, either by successfully returning from a `Thread.join()` method call or by a `Thread.isAlive()` method call returning `false` on that thread.
- *Thread interruption rule*: A thread calling the `Thread.interrupt()` method on another thread happens-before the interrupted thread detects the interrupt, either by having an `InterruptedException` thrown, or by invoking the `Thread .isInterrupted()` or the `Thread.interrupted()` method on itself.

Review Questions

22.4 Which of the following statements are guaranteed to be true about the following program?

Click here to view code image

```
public class ThreadedPrint {
  static Thread makeThread(String id, boolean daemon) {
    Thread t = new Thread(() -> System.out.println(id), id);
    t.setDaemon(daemon);
    t.start();
    return t;
  }

  public static void main(String[] args) {
    Thread a = makeThread("A", false);
    Thread b = makeThread("B", true);
    System.out.println("End");
```

```
  }
}
```

Select the two correct answers.

a. The letter `A` is always printed.

b. The letter `B` is always printed.

c. The letter `A` is never printed after `End`.

d. The letter `B` is never printed after `End`.

e. The program might print `B`, `End`, and `A`, in that order.

22.5 Which of the following statements are true about the `synchronized` statement? Select the two correct answers.

a. If the expression in a `synchronized` statement evaluates to `null`, a `NullPointer-Exception` will be thrown.

b. The lock is only released if the execution of the `synchronized` statement terminates normally.

c. Several `synchronized` statements, synchronizing on the same object, can be executed concurrently.

d. Synchronized statements cannot be nested.

e. The block notation, `{}`, cannot be omitted even if there is only a single statement to execute in a `synchronized` statement.

22.6 Given the following program, which code modifications will result in *both* threads being able to participate in printing one smiley face (`:-)`) on a line continuously?

Click here to view code image

```
public class Smiley extends Thread {

  public void run() {                                          // (1)
    while (true) {                                             // (2)
      try {                                                    // (3)
          System.out.print(":");                               // (4)
          sleep(100);                                          // (5)
          System.out.print("-");                               // (6)
          sleep(100);                                          // (7)
          System.out.println(")");                             // (8)
          sleep(100);                                          // (9)
```

```
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    }

    public static void main(String[] args) {
      new Smiley().start();
      new Smiley().start();
    }
  }
```

Select the two correct answers.

a. Synchronize the `run()` method at (1) with the keyword `synchronized`.

b. Synchronize the `while` loop at (2) with a `synchronized(Smiley.class)` statement.

c. Synchronize the `try-catch` construct at (3) with a `synchronized(Smiley.class)` statement.

d. Synchronize the statements from (4) to (9) with a `synchronized(Smiley.class)` statement.

e. Synchronize each statement at (4), (6), and (8) individually with a `synchronized (Smiley.class)` statement.

22.7 Which of the following events will cause a thread to terminate?

Select the one correct answer.

a. The method `sleep()` is called.

b. The method `wait()` is called.

c. Execution of the `start()` method ends.

d. Execution of the `run()` method ends.

e. Execution of the thread constructor ends.

22.8 Which of the following statements are true about the following code?

Click here to view code image

```
public class Joining {
  static Thread createThread(final int i, final Thread t1) {
    Thread t2 = new Thread() {
      public void run() {
```

```
          System.out.println(i+1);
          try {
            t1.join();
          } catch (InterruptedException ie) {
          }
          System.out.println(i+2);
        }
      };
      System.out.println(i+3);
      t2.start();
      System.out.println(i+4);
      return t2;
    }

    public static void main(String[] args) {
      createThread(10, createThread(20, Thread.currentThread()));
    }
  }
```

Select the two correct answers.

a. The first number printed is `13`.

b. The number `14` is printed before the number `22`.

c. The number `24` is printed before the number `21`.

d. The last number printed is `12`.

e. The number `11` is printed before the number `23`.

22.9 What can be guaranteed by calling the method `yield()`?

Select the one correct answer.

a. All lower-priority threads will be granted CPU time.

b. The current thread will sleep for some time while some other threads run.

c. The current thread will not continue until other threads have terminated.

d. The thread will wait until it is notified.

e. None of the above

22.10 In which class or interface is the `notify()` method defined?

Select the one correct answer.

a. `Thread`

b. `Object`

c. `Appendable`

d. `Runnable`

22.11 What will be the result of invoking the `wait()` method on an object without ensuring that the current thread holds the lock on the object?

Select the one correct answer.

a. The code will fail to compile.

b. Nothing special will happen.

c. An `IllegalMonitorStateException` will be thrown if the `wait()` method is called and the current thread does not hold the lock of the object.

d. The thread will be blocked until it gains the lock of the object.

22.12 What will be the result of compiling and running the following program?

Click here to view code image

```
public class Tank {
  private boolean isEmpty = true;

  public synchronized void emptying() {
    pause(true);
    isEmpty = !isEmpty;
    System.out.println("emptying");
    notify();
  }

  public synchronized void filling() {
    pause(false);
    isEmpty = !isEmpty;
    System.out.println("filling");
    notify();
  }

  private void pause(boolean flag) {
    while (flag ? isEmpty : !isEmpty) {
      try {
        wait();
      } catch (InterruptedException ie) {
        System.out.println(Thread.currentThread().getName() + " interrupted.");
      }
```

```
    }
  }

  public static void main(String[] args) {
    final Tank token = new Tank();
    new Thread(() -> {for(;;) token.emptying();}, "A").start();
    new Thread(() -> {for(;;) token.filling();}, "B").start();
  }
}
```

Select the one correct answer.

a. The program will compile and continue running once started, but will not print anything.

b. The program will compile and continue running once started, printing only the string `"emptying"`.

c. The program will compile and continue running once started, printing only the string `"filling"`.

d. The program will compile and continue running once started, always printing the string `"filling"` followed by the string `"emptying"`.

e. The program will compile and continue running once started, printing the strings `"filling"` and `"emptying"` in some order.