10 Threads

Exam objectives covered in this chapter	What you need to know
[10.1] Create and use the `Thread` class and the `Runnable` interface	How to use class `Thread` and the `Runnable` interface to create classes that can execute as a separate thread of execution.
[10.2] Manage and control thread lifecycle	What states a thread of execution can be in, and how it transitions from one state to another including the methods involved. What is or can't be guaranteed by each method. How the execution of a method depends on the thread scheduling by the underlying operating system.
[10.3] Synchronize thread access to shared data	How sharing of objects between threads can lead to inconsistent memory and invalid object state. How to make your classes thread safe.
[10.4] Identify code that may not execute correctly in a multithreaded environment	How to identify classes, methods, and variables that might not work as expected in a multithreaded environment. How to fix them.

Have you ever watched a movie with a group of friends, laughing, shouting, and sharing popcorn, nachos, or fries—all at the *same* time? If yes, you've already practiced *multithreading* and *concurrency*. You can compare these separate tasks—watching a movie, laughing, shouting, and so on—with multiple threads (multithreading), which you execute *concurrently* (at the same time). Can you imagine watching a movie with your friends, pausing it so that you can (only) laugh, followed by

(only) shouting, and so on? It sounds bizarre. Similarly, apart from executing multiple applications concurrently, multithreading facilitates optimal use of a computer's processing capabilities. Modeled around human behavior, multithreading is the need of the hour.

Multithreading is implemented by the underlying OS by dividing and allotting processor time to the running applications using multiple algorithms (round-robin, high priority first, preemptive, and so on). Applications can execute as single or multiple *processes*, which might run multiple *threads*. Java supports multithreading by enabling you to define and create separate threads of execution in your applications. Most Java Virtual Machine (JVM) implementations run as a single *process*. A process is a self-contained execution environment complete with its own private set of basic runtime resources, including memory. A process can create multiple threads, also known as *lightweight* processes. Creation of a thread requires fewer resources. Multiple threads share the resources of the process in which they're created, like memory and files. They also have their own exclusive working space, like stacks and PC registers.

"Every problem has a solution, and every solution has a problem"—this statement holds true for threads. Threads were created so that they could make the best use of a processor's time. An application can create multiple threads. When threads share objects among themselves, it can result in interleaving operations (*thread interference*) and reading inconsistent object values (*memory inconsistency*). You can protect shared data by synchronizing its access. But this can lead to other issues like thread contention through *deadlock*, *starvation*, and *livelock*.

NOTE Even if you don't create a multithreaded Java application, you can't ignore these concepts. Java technologies like swing in Core Java and Servlets in web applications and many popular Java frameworks implicitly multithread and call your code from multiple threads. Developing safe code that works and behaves well in a multithreaded environment demands knowledge about multithreading and synchronization.

Multithreading is a *big* topic and its coverage in this chapter is limited to the topics covered on the exam. This chapter also includes an introduction to a few threading concepts that you must know to understand the chapter contents. These introductory sections can be identified by the use of the term *warm-up* in section headings. You can skip these if you're familiar with these concepts.

For the exam, you must be able to clearly identify what is guaranteed and what isn't by threads. For example, a thread is guaranteed to execute its own code in the sequence in which it's defined. But you can't guarantee when a thread *exactly* starts, pauses, or resumes its execution. Make note of these points.

EXAM TIP For the exam, it's important to understand what threads guarantee and what they don't. Expect multiple questions on *expected*, *probable,* and *assured* thread behavior.

This chapter covers

- How to create threads using class `Thread` and the `Runnable` interface
- How to define and use `Thread` instances to create separate threads of execution
- How to manage and control the thread lifecycle
- How to protect data from concurrent access by threads
- How to identify code that might not execute correctly in a multithreaded environment

Let's get started with creating and using threads.

10.1 Create and use threads

[10.1] Create and use the Thread class and the Runnable interface

All nontrivial Java applications are multithreaded. A JVM supports multiple thread execution. Multiple threads can be supported by an underlying system by using multiple hardware processors, by time-slicing a single processor, or by time-slicing multiple processors. Java language supports multithreading. Class `Thread` and the `Runnable` interface can be used to define and start your own threads. In the next chapter on concurrency, we'll explore the `java.util.concurrent` package to execute your threads.

You'll often hear about thread objects and threads of execution in a multithreaded or concurrent application. Though related, a `Thread` instance and a *thread of execution* aren't the same. A `Thread` instance is a Java object. The implementation of Java threads is JVM-specific. A JVM implementation might even choose not to map Java threads to native threads at all! Creation of a thread of execution is passed to the OS by a JVM implementation. A thread of execution has its own set of program counter (PC) registers to store the next set of instructions to execute. It also has its own *stack* that stores *method frames* to store the state of a method invocation. The state of a method invocation includes the value of local variables, method parameters, method return values, exception handler parameters, and intermediate values. A process like JVM's can create multiple threads of execution to execute multiple tasks simultaneously. A `Thread` instance and a thread of execution are depicted in figure 10.1.

Have you ever wondered how many threads your Java application has? At least one—the *main* thread. A JVM starts execution of a Java application using a thread of execution: main. This thread executes code defined in method `main()` and can create other threads of execution.

NOTE The main thread is named 'main' by the JVM. Don't confuse it with the method `main()`.

In this section, you'll see how to define and start your own threads using Java classes that either extend class `Thread` or implement the `Runnable` interface. Let's get started with an example.

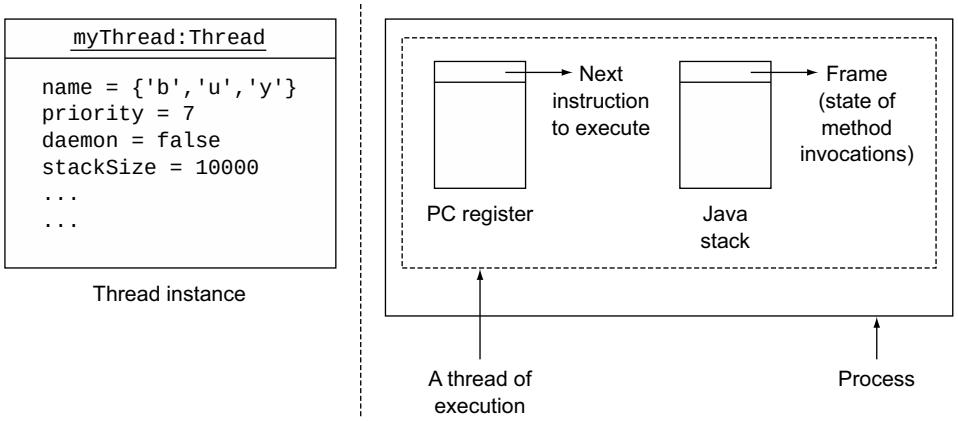

Figure 10.1 A `Thread` instance and a thread of execution

Have you tried dancing and singing simultaneously? Let's create a small program that emulates this behavior by creating and starting multiple threads simultaneously. Let's get started with extending class `Thread`.

10.1.1 Extending class Thread

Class `Thread` can be used to create and start a new thread of execution. To create your own thread objects using class `Thread`, you *must*

- Extend class `Thread`
- Override its method `run()`

To create and start a new thread of execution, you *must*

- Call `start()` on your `Thread` instance

When a new thread of execution starts, it will execute the code defined in the thread instance's method `run()`. Method `start()` will trigger creation of a new thread of execution, allocating resources to it. We'll cover the methods of class `Thread` as we move forward in the chapter.

The following listing shows the code for the sample application that creates a new thread `Sing` by extending class `Thread`, and instantiates this thread in method `main()`.

Listing 10.1 Using class `Thread` to create a thread

```
class Sing extends Thread{
    public void run() {
        System.out.println("Singing.......");
    }
}
```

run() begins execution when you call start() on a thread.

```
class SingAndDance {
    public static void main(String args[]) {
        Thread sing = new Sing();
        sing.start();
        System.out.println("Dancing");
    }
}
```

Instantiate Thread

Start thread sing

When you execute the code in listing 10.1, method `main()` might complete its execution before *or* after the thread `sing` starts its execution. The same code might generate different output when you execute it on the same or different systems, at different times. The exact time of thread execution is determined by how the underlying thread scheduler schedules the execution of threads. Because thread scheduling is specific to a specific JVM and depends on a lot of parameters, the same system might change the order of processing of your threads. The probable outputs of the code in listing 10.1 are shown in figure 10.2.

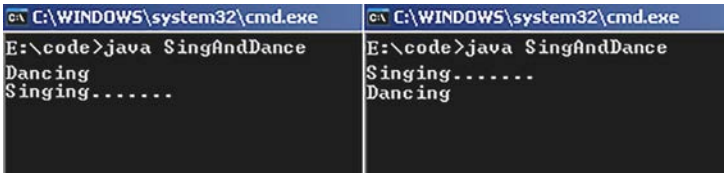

Figure 10.2 Probable outputs of class `SingAndDance`

EXAM TIP The exam might question you on the probable output of a multithreaded application. Because you can't be sure of the order of execution of threads by an underlying OS, such questions can have multiple correct code outputs.

In listing 10.1, note that calling `start()` on a `Thread` instance creates a new thread of execution. The default implementation of method `start()` in class `Thread` checks if you're starting it for the first time; if yes, it calls a native method to start a new thread of execution, calling its method `run()`. What happens if you override method `start()` without calling the superclass's default implementation?

```
class Sing extends Thread{
    public void run() {
        System.out.println("Singing.......");
    }
    public void start() {
        System.out.println("Starting...");
    }
}
class SingAndDance {
    public static void main(String args[]) {
        Thread sing = new Sing();
        sing.start();
        System.out.println("Dancing");
    }
}
```

Sing overrides start

Overridden start() doesn't start the thread; it won't create new thread of execution

The preceding code is guaranteed to output the following because it isn't starting another thread:

```
Starting...
Dancing
```

EXAM TIP Watch out for code that overrides method `start()` in a class that extends `Thread`. If it doesn't call method `start()` from its superclass `Thread`, it won't start a new thread of execution.

When you create a thread class by extending class `Thread`, you lose the flexibility of inheriting any other class. To get around this, instead of extending class `Thread`, you can implement the `Runnable` interface, as discussed next.

10.1.2 Implement interface Runnable

If a class implements the `Runnable` interface, its instances can define code that can be executed by threads of execution. The `Runnable` interface defines just one method, `run()`:

```
public interface Runnable {
    public abstract void run();
}
```

Steps for a class to use the `Runnable` interface

- Implement the `Runnable` interface
- Implement its method `run()`

Let's modify class `Sing` defined in listing 10.1 so that it implements the `Runnable` interface instead of extending class `Thread` (modifications are in bold):

Listing 10.2 Using the `Runnable` interface to create a thread

```
class Sing implements Runnable{
    @Override
    public void run() {
        System.out.println("Singing.......");
    }
}
class SingAndDance2 {
    public static void main(String args[]) {
        Thread sing = new Thread(new Sing());
        sing.start();
        System.out.println("Dancing");
    }
}
```

Sing implements Runnable

run begins execution when you call start on a thread.

Instantiate thread by passing instance of Runnable to Thread's constructor

Start thread sing.

NOTE While implementing or overriding methods, I won't use the annotation `@Override` in all code examples in the rest of the chapters because the code examples on the exam might not use it.

The preceding code will generate the same output as shown in figure 10.1. Apart from the most obvious difference that class `Sing` implements the `Runnable` interface, compare the difference in how `sing` is instantiated:

```
Thread sing = new Thread(new Sing());
```

If your class implements the `Runnable` interface, then you should pass its instance to the constructor of class `Thread`. If your class extends class `Thread`, then you can create a new thread by using the `new` operator and invoking its constructor.

Class `Thread` defines multiple overloaded constructors. The following constructors allocate a new `Thread` object:

```
Thread()
Thread(String name)
Thread(Runnable target)
Thread(Runnable target, String name)
```

A `Thread` constructor that accepts a string value enables you to create a thread with a name. But if you don't assign an explicit name to a thread, Java automatically generates and assigns a name to it. Make note of the `Thread` constructor that accepts a `Runnable` object. A `Thread` instance stores a reference to a `Runnable` object and uses it when you *start* its execution (by calling `start()`). Here's the partial code of class `Thread`:

```
public class Thread implements Runnable {
    /* What will be run. */
    private Runnable target;
    /*.. rest of the code.. */
}
```

Because class `Thread` implements the `Runnable` interface, you can instantiate a thread by passing it another `Thread` instance. So what happens when you create a thread, say, `A`, using another `Thread` instance, say, `B`? Does Java create multiple threads of execution in this case? When you execute `A.start()`, will it execute `A.run()` or `B.run()`? Let's answer all these questions using the next "Twist in the Tale" exercise.

Twist in the Tale 10.1

Class `SingAndDance` instantiates a thread by passing it an instance of the class that extends the thread itself. Will the following class execute method `start()` or `run()` twice? What do you think is the output of the following code?

```
class SingAndDance3 {
    public static void main(String args[]) {
        Thread sing = new Sing();
        Thread newThread = new Thread(sing);
        newThread.start();
    }
}
class Sing extends Thread{
    public void run() {
        System.out.println("Singing");
```

```
    }
}
```

a The code fails to compile.
b The code prints `Singing`.
c The code prints `Singing` twice.
d Class `SingAndDance3` starts one new thread of execution in `main`.
e Class `SingAndDance3` starts two new threads of execution in `main`.
f There is no output.

Each thread is created with a *priority*. Its range varies from 1 to 10, with 1 being the lowest priority and 10 the highest priority. By default, a thread creates another thread with the same priority as its own. It can also be explicitly set for a `Thread` instance, using `Thread`'s method `setPriority(int)`. A thread scheduler *might* choose to execute threads with higher priorities over threads with lower priorities, though you can't guarantee it.

EXAM TIP You can't guarantee that a thread with a higher priority will always execute before a thread with a lower priority.

When you call `start()` on a `Thread` instance, it creates a new thread of execution. A thread of execution has a lifecycle: a thread is created, it *might* be paused, and eventually it completes its execution. The next section explores the different states in which a thread can exist, the methods that you can call on a thread, and how these methods affect the states of a thread.

10.2 Thread lifecycle

[10.2] Manage and control thread lifecycle

A thread's lifecycle consists of multiple states. You can control the transition of a thread from one state to another by calling its methods. The methods signal the JVM regarding the change in status of a thread. The *exact time* of state transition is controlled by a thread scheduler, which is bound to vary across platforms.

10.2.1 Lifecycle of a thread

A thread can exist in multiple states: `NEW`, `RUNNABLE`, `WAIT`, `TIMED_WAITING`, `BLOCKED`, or `TERMINATED`. A thread doesn't begin its execution with its instantiation. Figure 10.3 shows the various phases of a thread, including the methods that can be called on a thread object in each state, which makes it transition from one state to another. For the exam, you must know how a thread transitions from one state to another and what methods you can call on a thread when it's in different phases.

A Java thread can officially exist in the states as defined by a thread's inner enum, `State`. Table 10.1 lists the values of enum `Thread.State`.

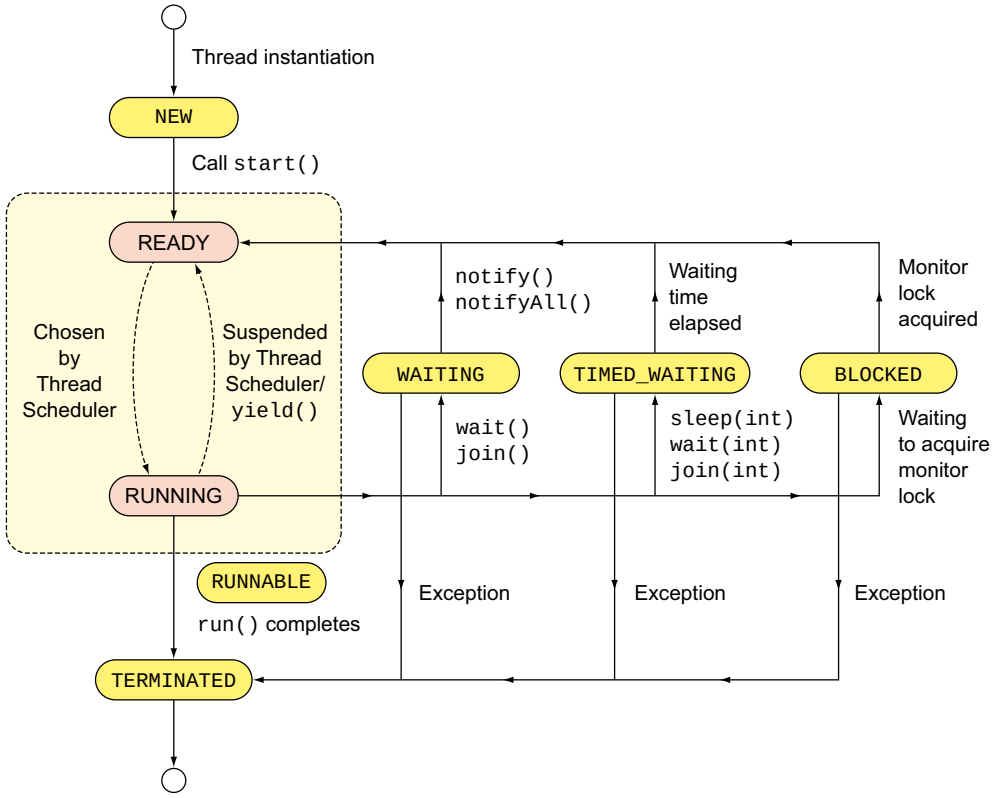

Figure 10.3 Various phases of a thread

Table 10.1 States of a thread as defined by enum `Thread.State`

Thread state	Description
NEW	A thread that has been created but hasn't yet started is in this state.
RUNNABLE	Thread state for a runnable thread. A thread in this state is executing in the JVM but it may be waiting for other resources from the OS, such as a processor.
BLOCKED	A thread that's blocked waiting for a monitor lock is in this state.
WAITING	A thread that's waiting indefinitely for another thread to perform a particular action is in this state.
TIMED_WAITING	A thread that's waiting for another thread to perform an action for up to a specified waiting time is in this state.
TERMINATED	A thread whose `run()` method has finished is in this state (still a thread object but not a thread of execution).

Calling `start()` on a new thread instance implicitly calls its `run()`, which transitions its state from `NEW` to `RUNNABLE`. A thread in the `RUNNABLE` state is all set to be executed. It's just waiting to be chosen by the thread scheduler so that it gets the processor time. Thread scheduling is specific to the underlying OS on every system. As a programmer, you can't control or determine when a particular thread transitions from the `READY` state to the `RUNNING` state, and when it actually gets to execute. A thread scheduler follows various scheduling mechanisms to utilize a processor efficiently, and also to give a fair share of processor time to each thread. It might suspend a running thread to give way to other READY threads and it might execute it later. The READY and the RUNNING states are together referred to as the `RUNNABLE` state.

EXAM TIP The states READY and RUNNING are together referred to as the `RUNNABLE` state.

A running thread enters the `TIMED_WAITING` state, when it might need to wait for a specified interval of time, before it can resume its execution. It happens when `sleep(int)`, `join(int)`, or `wait(int)` is called on a running thread. On completion of the elapsed interval, a thread enters the queue eligible to be scheduled by the thread scheduler. When `join()` or `wait()` is called on a running thread, it transitions to the `WAITING` state. It can change back to the `RUNNABLE` state when `notify()` or `notifyAll()` is called (the details of all these methods are discussed later). A `RUNNABLE` thread might enter the `BLOCKED` state when it's waiting for other system resources like network connections or to acquire an object lock to execute a synchronized method or code block. Depending on whether the thread is able to acquire the monitor lock or resources, it returns back to the `RUNNABLE` state.

With the successful completion of `run()`, a thread enters the `TERMINATED` state. A thread might transition from any state to the `TERMINATED` state due to an exception. You can execute the following quick code to get a list of all the threads that are active on your system and the state they're in:

```
Set<Thread> threadSet = Thread.getAllStackTraces().keySet();
for(Thread t : threadSet)
    System.out.println(t + " --- " + t.getState());
```

Here's probable output of this code (which might vary on each individual's system):

```
Thread[Reference Handler,10,system] --- WAITING
Thread[Finalizer,8,system] --- WAITING
Thread[Attach Listener,5,system] --- RUNNABLE
Thread[Signal Dispatcher,9,system] --- RUNNABLE
Thread[Thread-1,5,main] --- TIMED_WAITING
```

It's very likely for the exam to question you on what happens if you call a method (and how many times) that you weren't supposed to in a particular thread state. For example, can you call `join()` on a thread in its `NEW` state, or, say, can you call `start()`

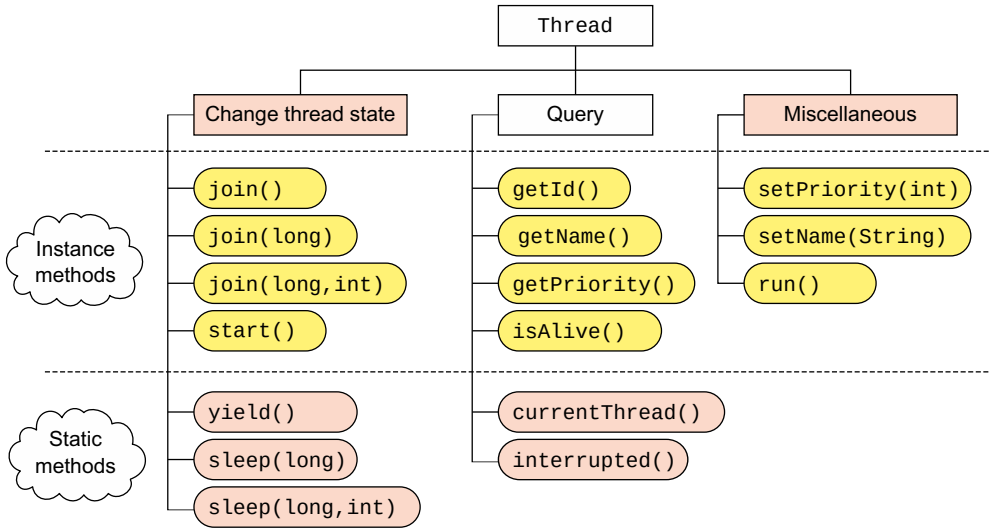

Figure 10.4 Main methods of class `Thread`

on a thread twice? To answer these questions, you'll need to know the thread methods, covered next.

10.2.2 Methods of class Thread

Before we can move forward with the methods that can be called in a particular thread state, let's categorize the methods of class `Thread` as instance or static methods, and if these methods can be used to change a thread's state or to query it. The thread methods are shown in figure 10.4.

NOTE Class `Thread` includes deprecated methods, like `resume()`, `stop()`, and `suspend()`. Because their use isn't encouraged, they aren't discussed in this chapter. You won't find the deprecated methods on the exam either.

The next section dives into the methods that you can call to transition a thread in the `NEW` state to a thread in the `RUNNABLE` state.

10.2.3 Start thread execution

Calling `start()` on a `Thread` instance creates a new thread of execution, which executes `run()`. You can call `start()` on a thread that's in the `NEW` state. Calling `start()` from any other thread state will throw an `IllegalThreadStateException`:

```
class CantCallStartOnSameThreadMoreThanOnce {
    public static void main(String args[]) {
        Thread sing = new Sing();
```

State of thread sing is NEW

```
        sing.start();
        sing.start();
    }
}
class Sing extends Thread{
    public void run() {
        System.out.println("Singing");
    }
}
```

Calls start()—state of thread sing changes to RUNNABLE.

Can't call start() on a thread state other than NEW; throws java.lang.IllegalThreadStateException.

The preceding code might print `Singing` and then throw an `IllegalThreadStateException`. It might throw an `IllegalThreadStateException` without printing `Singing`. How? The preceding code might start the thread `sing`, but *before* it prints `Singing`, method `main()` might call `start()` on the `sing` thread again, throwing an `IllegalThreadStateException`.

EXAM TIP You can call `start()` only once on a `Thread` instance when it's in the `NEW` state. Calling `start()` on a thread in any other state will throw an `IllegalThreadStateException`.

What happens if you replace `sing.start()` with `sing.run()`?

```
class CanCallRunMultipleTimes {
    public static void main(String args[]) {
        Thread sing = new Sing();
        sing.run();
        sing.run();
    }
}
class Sing extends Thread{
    public void run() {
        System.out.println("Singing");
    }
}
```

State of thread sing—NEW

Calls run()—State of thread remains NEW.

The output of the preceding code is as follows:

```
Singing
Singing
```

In the preceding code, `run()` neither starts a new thread of execution in `main`, nor modifies the state of thread `sing` ❶. The `main` calls `run` like any other method—waiting for `run` to complete execution, before executing the code at ❷. The code at ❷ again executes `run()` and prints `Singing`.

EXAM TIP Calling `run()` on a `Thread` instance doesn't start a *new* thread of execution. The `run()` continues to execute in the *same* thread. Watch out for trick questions on using `start()` versus `run()` in the exam.

Because `start()` starts a new thread of execution, you might find it interesting how the exceptions are handled in multithreaded applications. For example, if both the

calling thread and the called thread throw unhandled exceptions at runtime, do you think the code will encounter only one or both? Let's find out using the next "Twist in the Tale" exercise.

Twist in the Tale 10.2

What do you think is the output of the following code?

```
class Twist10_2 {
    public static void main(String args[]) {
        Thread sing = new Sing();
        sing.start();
        throw new RuntimeException("main");
    }
}
class Sing extends Thread{
    public void run() {
        System.out.println("Singing");
        throw new RuntimeException("run");
    }
}
```

a
```
java.lang.RuntimeException: main
Singing
java.lang.RuntimeException: run
```

b
```
Singing
java.lang.RuntimeException: run
java.lang.RuntimeException: main
```

c
```
java.lang.RuntimeException: main
```

d
```
Singing
java.lang.RuntimeException: run
```

e
```
Singing
java.lang.RuntimeException: main
java.lang.RuntimeException: run
```

Once a thread begins its execution, multiple factors might pause its execution. Let's examine them in detail in the next section.

10.2.4 Pause thread execution

A thread might pause its execution due to the calling of an explicit method or when its time slice with the processor expires.

THREAD SCHEDULING

A processor's time is usually time-sliced to allow multiple threads to run, with each thread using one or more time slices. A thread scheduler might suspend a running thread and move it to the ready state, executing another thread from the ready queue. Thread scheduling is specific to JVM implementation and beyond the control of an

application programmer. The scheduler moves a thread from the `READY` state to `RUNNING` and vice versa, to support concurrent processing of threads.

EXAM TIP As a Java programmer, you can't control or determine when the thread scheduler moves a thread from the `RUNNING` state to `READY` and vice versa. It's specific to an OS.

METHOD THREAD.YIELD()

Imagine while debugging or testing your code, you need to reproduce a bug due to race conditions (that is, when multiple threads compete). You can insert a call to `Thread.yield()` in one of the threads. The static method `yield()` makes the currently executing thread pause its execution and give up its current use of the processor. But it only acts as a hint to the scheduler. The scheduler might also ignore it. The static method `yield()` can be placed literally anywhere in your code—not only in method `run()`:

```
class YieldProcessorTime {
    public static void main(String args[]) {
        Thread sing = new Sing();
        sing.start();
        Thread.yield();
    }
}
class Sing extends Thread{
    public void run() {
        yield();
        System.out.println("Singing");
    }
}
```

Might cause thread main to yield its processor time.

When executed, might cause thread sing to yield its processor time.

As shown in figure 10.5, when called from two threads, thread 1 and thread 2, `yield()` might or might not *yield* its execution.

So what's guaranteed from this method call? To be precise, nothing. It might not make the currently executing thread give up its processor time. If it does, it doesn't guarantee when it will happen and when the thread will resume its execution.

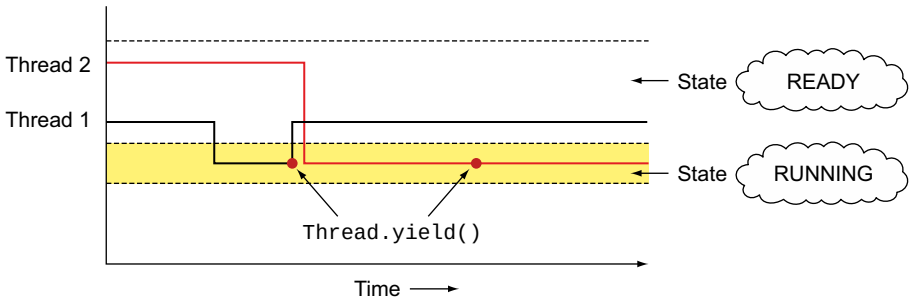

Figure 10.5 Calling `yield()` might yield the current thread's execution slot as soon as it's called or after a delay, or it might be ignored.

EXAM TIP Method `yield()` is static. It can be called from any method, and it doesn't throw any exceptions.

METHOD THREAD.SLEEP()

The static method `Thread.sleep()` is *guaranteed* to cause the currently executing thread to temporarily give up its execution for *at least* the specified number of milliseconds (and nanoseconds) and move to the `READY` state. You use `sleep()` to *slow down* the execution of your thread. Imagine you're using a thread to animate a ball across a visible frame. Execute the following class and you'll see that a black ball zooms across the screen:

```
import javax.swing.*;
import java.awt.*;
class MyFrame {
    public static void main(String args[]) {
        JFrame frame = new JFrame();
        frame.setSize(400, 300);
        frame.setVisible(true);
        MovingBall ball = new MovingBall(60, frame);
        ball.start();
    }
}
class MovingBall extends Thread{
    int radius;
    Graphics g;
    int xPos, yPos;
    JFrame frame;
    MovingBall(int radius, JFrame frame) {
        this.radius = radius;
        this.g = frame.getGraphics();
        this.frame = frame;
    }
    public void run() {
        while (true) {
            g.setColor(Color.WHITE);
            g.fillRect(0, 0, frame.getWidth(), frame.getHeight());
            ++xPos; ++yPos;
            g.setColor(Color.BLACK);
            g.fillOval(xPos, yPos, radius, radius);
        }
    }
}
```

Create a frame.

Create and start MovingBall thread.

x and y positions to draw an oval

Execute it forever.

Paint whole screen white

Modify x and y position of moving ball

Draw ball at new x and y position

In the preceding code, method `run()` tries to animate a ball by first painting the complete visible frame area with white and then drawing a black oval at the specified `x` and `y` positions. Each loop repeats these steps with modified `x` and `y` coordinates that make the ball move across the frame. You can slow down the moving ball by making `MovingBall` sleep for the specified milliseconds (modifications are in bold):

```
class MovingBall extends Thread{
    int radius;
    Graphics g;
```

```
    int xPos, yPos;
    JFrame frame;
    MovingBall(int radius, JFrame frame) {
        this.radius = radius;
        this.g = frame.getGraphics();
        this.frame = frame;
    }
    public void run() {
        while (true) {
            try {
                Thread.sleep(10);
            }
            catch (InterruptedException e) {
                System.out.println(e);
            }
            xPos = xPos + 2; yPos = yPos + 2;
            g.setColor(Color.WHITE);
            g.fillRect(0, 0, frame.getWidth(), frame.getHeight());
            g.setColor(Color.BLACK);
            g.fillOval(xPos, yPos, radius, radius);
        }
    }
}
```

Guarantees to sleep for at least 10 milliseconds (if not interrupted).

If interrupted, sleeping thread might throw InterruptedException

Class `Thread` defines the overloaded versions of `sleep()` as follows:

```
public static native void sleep(long milli) throws InterruptedException;
public static void sleep(long milli, int nanos) throws InterruptedException
```

Whether a thread will sleep for the precise duration specified in nanoseconds will depend on an underlying system. Unless interrupted, the currently executing thread will sleep at least for the specified duration. On the exam, watch out for questions that state a thread will become runnable *exactly* after the expiration of the sleep duration. Unless interrupted, a thread is guaranteed to sleep for *at least* the specified duration. The exact time to resume execution depends on the thread scheduler.

EXAM TIP A thread that's suspended due to a call to `sleep` doesn't lose ownership of any monitors.

You can also expect questions on where to place a call to method `sleep()`. The answer depends on who is executing a call to `sleep()`. Like `yield()`, `sleep()` is `Thread`'s static method. It makes the currently executing thread give up its execution and sleep. It can be called from any piece of code—all code is executed by some thread. If it's placed in `Runnable`'s `run()`, it will cause the thread to sleep. Placed otherwise, it will make the calling thread sleep. What happens if you place `sleep()` in `MovingBall`'s constructor (showing only relevant code)?

```
class MovingBall extends Thread{
    //..code not shown deliberately
    MovingBall(int radius, JFrame frame) {
```

```
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException e) {
                System.out.println(e);
            }
            //..code not shown deliberately
        }
        public void run() {
            //..code not shown deliberately
        }
}
```

A thread that instantiates `MovingBall` will execute `sleep()`, and then sleep for the specified duration before it can complete `MovingBall`'s instantiation.

METHOD JOIN()

A thread might need to pause its *own* execution when it's waiting for another thread to complete its task. If thread A calls `join()` on a `Thread` instance B, A will wait for B to complete its execution before A can proceed to its own completion. Imagine multiple teams—design, development, and testing—are working on a software project. The project can't be sent to a customer until all of these teams complete their tasks. Ideally, the delivery process will wait for design development and testing teams to complete their tasks before the delivery process can move ahead with its own task of handing over the project to a customer. Let's code a subset of this example. The design team must complete the design of a screen before a developer can start coding it:

```
class ScreenDesign extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) System.out.println(i);
    }
}
class Developer {
    ScreenDesign design;
    Developer(ScreenDesign design) {
        this.design = design;
    }
    public void code() {
        try {
            System.out.println("Waiting for design to complete");
            design.join();
            System.out.println("Coding phase start");
        }
        catch(InterruptedException e) {
            System.out.println(e);
        }
    }
}
class Project {
    public static void main(String[] args) {
        ScreenDesign design = new ScreenDesign(); design.start();
```

run() simply prints a couple of numbers.

Developer stores a reference to Design.

code() calls design.join().

join() throws checked InterruptedException.

Start thread design.

```
        Developer dev = new Developer(design);
        dev.code();                                  <-- dev.code(), which runs in thread main, calls design.join.
    }
}
```

Figure 10.6 shows probable outputs of the preceding code. Because class `Developer` isn't a `Runnable` instance, it doesn't execute it in its own thread of execution; rather, it executes in the thread `main`. When `Developer` calls `design.join()`, the thread `main` waits for `design` to complete its execution before executing the rest of its code.

0	0	Waiting for design to complete
Waiting for design to complete	1	0
1	2	1
2	3	2
3	Waiting for design to complete	3
4	4	4
Coding phase start	Coding phase start	Coding phase start

Figure 10.6 Probable outputs of calling `join()` on a thread

EXAM TIP Method `join()` guarantees that the calling thread won't execute its remaining code until the thread on which it calls `join()` completes.

Class `Thread` defines overloaded `join()` methods as follows:

```
public final synchronized void join(long milli) throws InterruptedException
public final synchronized void join(long millis, int nanos)
                                                  throws InterruptedException
public final void join() throws InterruptedException
```

The variations of `join()` that accept milliseconds and nanoseconds wait for at least the specified duration (if they're not interrupted!). Behind the scenes, `join()` is implemented using methods `wait()`, `isAlive()`, and `notifyAll()`.

METHODS WAIT(), NOTIFY(), AND NOTIFYALL()

Imagine a server accepts and queues multiple requests from users that are processed by a thread. This thread might need to wait and pause its own execution if there are no new requests in the queue. A thread can pause its execution and wait on an object, a queue in this case, by calling `wait()`, until another thread calls `notify()` or `notifyAll()` on the same object.

Methods `wait()`, `notify()`, and `notifyAll()` can be called on all Java objects, because they're defined in class `Object` and not class `Thread`. Because these methods can be invoked by synchronized methods and blocks and by threads that own the object monitors, we'll cover these methods in detail in the next section after covering object monitors and synchronized methods and blocks.

10.2.5 End thread execution

A thread completes its execution when its method `run()` completes. You must not call the deprecated method `stop()` to stop execution of a thread. A thread might perform a task for a finite number of times or it might loop for an indefinite number of times. In the later case, the thread should define an exit condition to indicate its completion:

```
class Sing extends Thread{
    boolean singStatus = true;
    public void run() {
        while (singStatus)
            System.out.println("Singing");
    }
    public void setSingStatus(boolean value) {
        singStatus = value;
    }
}
```

run() executes until singStatus is false.

In the preceding code, the instance of thread `Sing` will execute until its instance variable `singStatus` is assigned a value of `false`. An application can modify the value of `singStatus` using `setSingStatus(boolean)`. Once `run()` completes its execution, the status of a thread changes to `TERMINATED`.

You create threads to speed up execution of a process so that unrelated tasks don't need to be executed in sequence and can execute concurrently. But these multiple threads might share data among themselves. Next, let's see how sharing data among multiple threads can result in incorrect data, and how to protect shared data among threads.

10.3 Protect shared data

[10.3] Synchronize thread access to shared data

Threads are lightweight processes that share certain areas of the memory, unlike regular processes. This makes threads very efficient, but it also introduces additional complexities regarding memory management. You can protect your shared data by making it accessible (for reading and writing) to only one thread at a point of time. Other techniques include defining *immutable* classes (the states of which can't be modified) and defining volatile variables. In this section, you'll identify the data that's shared across threads and needs to be protected, and the related techniques.

Let's get started with the identification of what data can be shared among multiple threads.

10.3.1 Identifying shared data: WARM-UP

A JVM instance hosts only one Java application. A Java application can create and execute multiple threads. When a new thread is created, it's allotted its exclusive share in the memory. The JVM also allows partial runtime data to be shared between

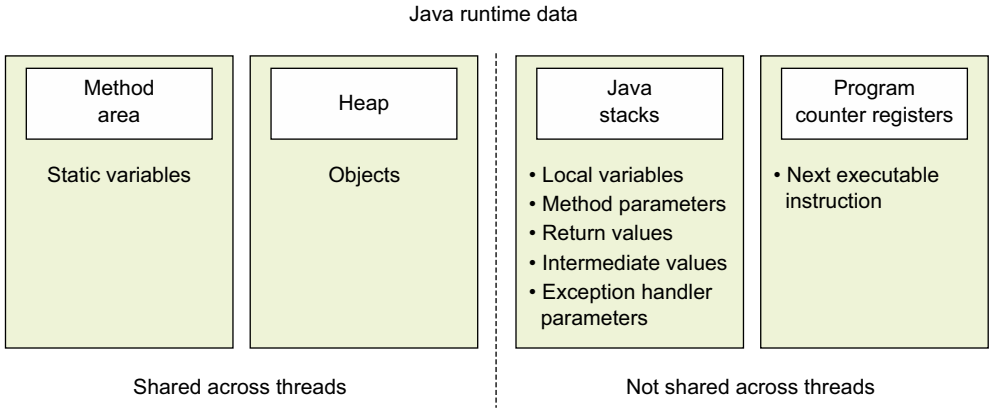

Figure 10.7 Identifying parts of runtime data that can be shared across threads

threads. A JVM's runtime data set includes the method area, the heap, Java stacks, and PC registers (it also includes native method stacks, which aren't covered by the exam, so they aren't discussed here). The method area includes class information that includes a note on all its variables and methods. This data set includes the static variables, which are accessible and shared by all the threads in a JVM. The static variables are shared by one class loader—that is, each class loader has its own set of static variables. The heap includes objects that are created during the lifetime of an application, again shared by multiple threads and processes. PC registers and Java stacks aren't shared across the threads. On instantiation, each thread gets a PC register, which defines the next set of instructions to execute. Each thread is also assigned a Java stack. When a thread begins execution of a method, a method frame is created and inserted into the java stack, which is used to store the local variables, method parameters, return values, and intermediate values that might be used in a method. Of all this data, the method area and heap are shared across threads. Figure 10.7 shows this arrangement.

In the next section, you'll see how multiple threads can interfere with each other's working.

10.3.2 Thread interference

Interleaving of multiple threads that manipulate shared data using multiple steps leads to thread interference. You can't assume that a single statement of Java code executes atomically as a single step by the processor. For example, a simple statement like incrementing a variable value might involve multiple steps like loading of the variable value from memory to registers (working space), incrementing the value, and reloading the new value in the memory. When multiple threads execute this seemingly atomic statement, they might interleave, resulting in incorrect variable values.

NOTE Operations that use arithmetic and assignment operators like `++`, `--`, `+=`, `-=`, `*=`, and `/=` aren't atomic. Multiple threads that manipulate variable values using these operators can interleave.

Let's work with an example of a class `Book`, which defines an instance variable `copiesSold` that can be manipulated using its methods `newSale()` or `returnBook()`:

```
class Book{
    String title;
    int copiesSold = 0;
    Book(String title) {
        this.title = title;
    }
    public void newSale() {
        ++copiesSold;
    }
    public void returnBook() {
        --copiesSold;
    }
}
```

Nonatomic statements include loading variable values from memory to registers, manipulating values, and loading them back to memory.

The threads `OnlineBuy` and `OnlineReturn` manipulate the value of class `Book`'s instance variable `copiesSold` in their `run()` using methods `newSale()` and `returnBook()`:

```
class OnlineBuy extends Thread{
    private Book book;
    public OnlineBuy(Book book) {
        this.book = book;
    }
    @Override
    public void run() {
        book.newSale();
    }
}
class OnlineReturn extends Thread{
    private Book book;
    public OnlineReturn(Book book) {
        this.book = book;
    }
    @Override
    public void run() {
        book.returnBook();
    }
}
```

When started, OnlineBuy calls newSale() on its Book instance.

When started, OnlineReturn calls returnBook() on its Book instance.

Let's see what happens when another class, say, `ShoppingCart`, instantiates a `Book` and passes it to the threads `OnlineBuy` and `OnlineReturn`:

Instantiate Book instance book.

```
class ShoppingCart {
    public static void main(String args[]) throws Exception {
        Book book = new Book("Java");
        Thread task1 = new OnlineBuy(book); task1.start();
```

Instantiate OnlineBuy using book and start thread task1.

```
        Thread task2 = new OnlineBuy(book); task2.start();
        Thread task3 = new OnlineReturn(book); task3.start();
    }
}
```

Instantiate another OnlineBuy instance using book and start thread task2.

Instantiate OnlineReturn instance using book and start thread task3.

In the preceding code method `main()` starts three threads—`task1`, `task2`, and `task3`. These threads manipulate and share the same `Book` instance, `book`. The threads `task1` and `task2` execute `book.newSale()`, and `task3` executes `book.returnBook()`. As mentioned previously, `++copiesSold` and `--copiesSold` aren't atomic operations. Also, as a programmer you can't determine or command the exact time when these threads will start with their execution (it depends on how they're scheduled to execute by the OS). Let's assume that `task2` starts with its execution and reads `book.copiesSold`. Before it can modify this shared value, it's also read by threads `task3` and `task2`, which are unaware that this value is being modified by another thread. All these threads modify the value of `book.copiesSold` and update it back in order. The last thread to update the value `book.copiesSold` overrides updates of the other two threads. Figure 10.8 shows one of the possible ways in which the threads `task1`, `task2`, and `task3` can interleave.

So, how can you assure that when multiple threads update shared values it doesn't lead to incorrect results? How can you communicate between threads? Let's discuss this in the next section.

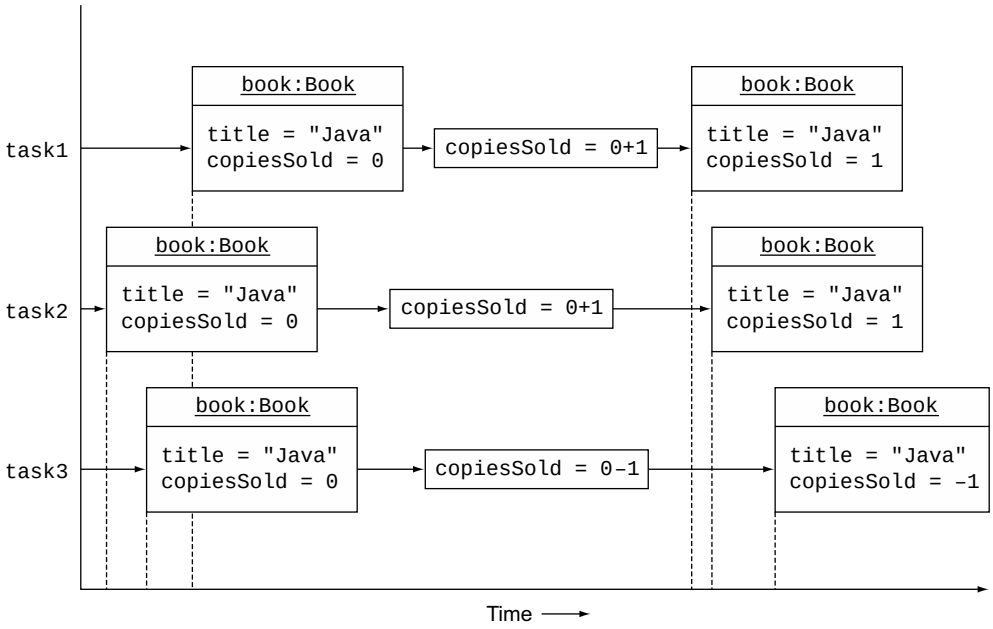

Figure 10.8 Interleaving threads can lead to incorrect results.

10.3.3 Thread-safe access to shared data

Imagine if you could lock a shared object while it was being accessed by a thread; you'd be able to prevent other threads from modifying it. This is exactly what Java does to make its data *thread safe*. Making your applications thread safe means securing your shared data so that it stores correct data, even when it's accessed by multiple threads. Thread safety isn't about *safe threads*—it's about safeguarding your shared data that might be accessible to multiple threads. A thread-safe class stores correct data without requiring calling classes to guard it.

You can lock objects by defining *synchronized methods* and *synchronized statements*. Java implements synchronization by using monitors, covered in the next quick warm-up section.

OBJECT LOCKS AND MONITORS: WARM-UP

Every Java object is associated with a *monitor*, which can be *locked* or *unlocked* by a thread. At a time, only one thread can hold lock on a monitor and is referred to as the *monitor owner*. Owning the monitor is also referred to as *acquiring the monitor*. If another thread wants to acquire the monitor of that object, it must wait for it to be released.

When the keyword `synchronized` is added to a method, only one thread can execute it at a time. To execute a synchronized method, a thread *must* acquire the monitor of the object on which the method is called. When the monitor of an object is locked, no other thread can execute any synchronized method on that object. When acquired, a thread can release the lock on the monitor if

- It has completed its execution.
- It needs to wait for another operation to complete.

For the first case, it releases the lock and exits. For the latter case, it enters a *waiting set* of threads that are waiting to acquire the monitor again. Threads enter the waiting set due to an execution of yield or wait. They can reacquire the monitor when `notifyAll()` or `notify()` is called.

Figure 10.9 shows how a thread might have to wait to hold a lock on a monitor. Only one thread can own the monitor of an object, and a thread might release the monitor and enter a waiting state.

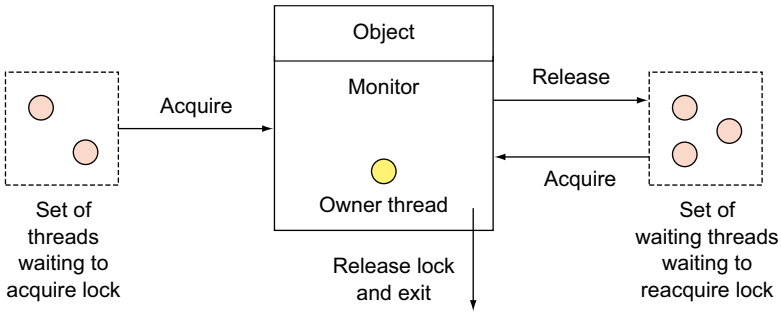

Figure 10.9 A thread must own a monitor before it can execute synchronized code. Only the monitor owner can execute the synchronized code.

With an understanding of how objects and their monitors, threads, and execution of synchronized code work together, let's modify the code used in section 10.3.2 so that threads working with shared data don't interleave.

SYNCHRONIZED METHODS

Synchronized methods are defined by prefixing the definition of a method with the keyword `synchronized`. You can define both instance and static methods as synchronized methods. The methods, which modify the state of instance or static variables, should be defined as synchronized methods. This prevents multiple threads from modifying the shared data in a manner that leads to incorrect values.

When a thread invokes a synchronized method, it automatically locks the monitor. If the method is an instance method, the thread locks the monitor associated with the instance on which it's invoked (referred to as `this` within the method). For static methods, the thread locks the monitor associated with the `Class` object, thereby representing the class in which the method is defined. These locks are released once execution of the synchronized method completes, with or without an exception.

EXAM TIP For nonstatic instance synchronized methods, a thread locks the monitor of the object on which the synchronized method is called. To execute static synchronized methods, a thread locks the monitor associated with the `Class` object of its class.

In the following listing, let's modify the definition of class `Book` by defining its methods `newSale()` and `returnBook()` as synchronized methods. The methods that belong to the data being protected are defined as synchronized.

Listing 10.3 Working with synchronized methods

```
class Book{
    String title;
    int copiesSold = 0;
    Book(String title) {
        this.title = title;
    }
    synchronized public void newSale() {
        ++copiesSold;
    }
    synchronized public void returnBook() {
        --copiesSold;
    }
}
class OnlineBuy extends Thread{
    private Book book;
    public OnlineBuy(Book book) {
        this.book = book;
    }
```

newSale() is now a synchronized method.

returnBook() is now a synchronized method.

```
    @Override
    public void run() {
        book.newSale();
    }
}
class OnlineReturn extends Thread{
    private Book book;
    public OnlineReturn(Book book) {
        this.book = book;
    }
    @Override
    public void run() {
        book.returnBook();
    }
}
class ShoppingCart {
    public static void main(String args[]) throws Exception {
        Book book = new Book("Java");
        Thread task1 = new OnlineBuy(book); task1.start();
        Thread task2 = new OnlineBuy(book); task2.start();
        Thread task3 = new OnlineReturn(book); task3.start();
    }
}
```

Figure 10.10 shows how threads `task2`, `task1`, and `task3` might acquire a lock on the monitor of object `book` defined in the `main` method of class `ShoppingCart`. As you see, a thread can't execute synchronized methods `newSale()` and `returnBook()` on `book` without acquiring a lock on its monitor. So each thread has exclusive access to `book` to modify its state, resulting in a *correct* book state at the completion of `main`.

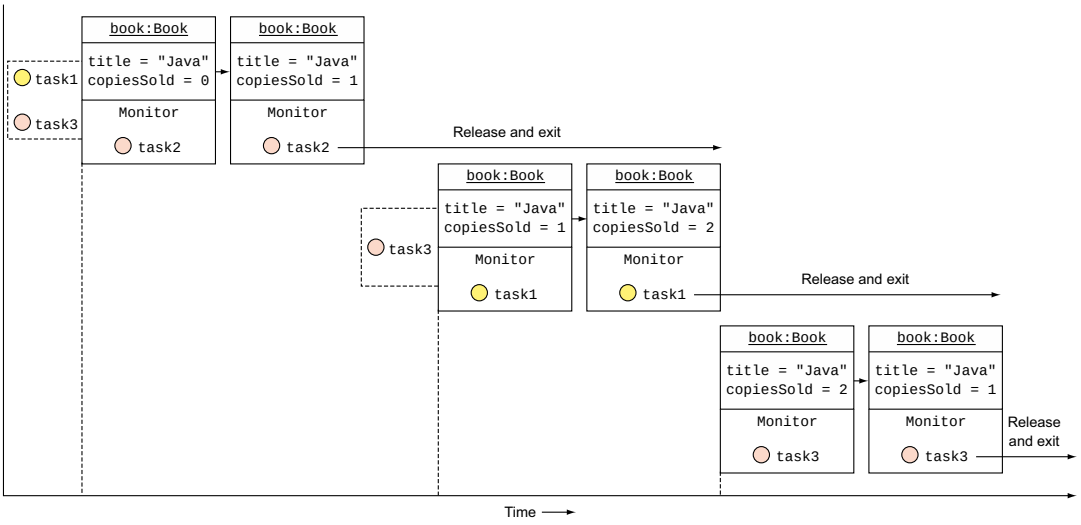

Figure 10.10 Threads acquire a lock on the monitor of object `book` before executing its synchronized methods `newSale()` and `returnBook()`. So threads `task1`, `task2`, and `task3` don't interleave.

EXAM TIP A thread releases the lock on an object monitor after it exits a synchronized method, whether due to successful completion or due to an exception.

What happens if, instead of defining methods `newSale()` and `returnBook()` as synchronized methods (see listing 10.3), you define the `run()` methods in the threads `OnlineBuy` and `OnlineReturn` as synchronized methods? Will this modification protect the data of shared object `book` in `ShoppingCart`? Let's answer this question in the next "Twist in the Tale" exercise. Take a closer look; apart from the mentioned modification, I've also modified some other bits of code.

Twist in the Tale 10.3

What do you think is the output of the following code?

```
class Book{
    int copiesSold = 0;
    public void newSale() { ++copiesSold; }
    public void returnBook() { --copiesSold; }
}
class OnlineBuy extends Thread{
    private Book book;
    public OnlineBuy(Book book) { this.book = book; }
    synchronized public void run() {                    //1
        book.newSale();
    }
}
class OnlineReturn extends Thread{
    private Book book;
    public OnlineReturn(Book book) { this.book = book; }
    synchronized public void run() {                    //2
        book.returnBook();
    }
}
class ShoppingCart {
    public static void main(String args[]) throws Exception {
        Book book = new Book();                         //3
        Thread task1 = new OnlineBuy(book); task1.start();     //4
        Thread task2 = new OnlineBuy(book); task2.start();     //5
        Thread task3 = new OnlineReturn(book); task3.start();  //6
    }
}
```

a The code on lines 1 and 2 fails to compile. It can't override `run` by adding the keyword `synchronized`.

b The code compiles, but calling `start()` on lines 4, 5, and 6 doesn't call `run` defined on lines 1 and 2. It calls the default `run()` method defined in class `Thread`.

c Lines 4, 5, and 6 call `run()` defined at lines 1 and 2, but fail to protect data of object `book` defined at line 3.

d Lines 4, 5, and 6 call `run()` defined at lines 1 and 2, and succeed in protecting data of object `book` defined at line 3.

As stated before, when a thread acquires a lock on the object monitor, no other thread can execute any other synchronized method on the object until the lock is released. This could become inefficient if your class defined synchronized methods that manipulate different sets of unrelated data. To do so, you might mark a block of statements with the keyword `synchronized`, as covered in the next section.

SYNCHRONIZED STATEMENTS

To execute synchronized statements, a thread must acquire a lock on an object monitor. For an instance method, it might not acquire a lock on the instance itself. You can specify *any* object on which a thread *must* acquire the monitor lock before it can execute the synchronized statements.

In the previous example for class `Book`, let's remove the keyword `synchronized` from the definition of method `newSale()` and define synchronized statements in it:

```
class Book{
    double rank = 0;
    int copiesSold = 0;

    Object objSale = new Object();
    Object objPos = new Object();

    public void newSale() {
        synchronized(objSale) {
            ++copiesSold;
        }
    }
    public void rankImprove() {
        synchronized(objPos) {
            --rank;
        }
    }
}
```

❶ Threads acquire lock on monitor of these objects to execute synchronized statements in newSale() and rankImprove()

❷ newSale() isn't a synchronized method.

❸ Start of synchronized block in newSale()

❹ End of synchronized block in newSale()

❺ Instance method that includes a synchronized statement that manipulates instance variable rank

The code at ❶ defines objects `objSale` and `objPos` that are used to execute synchronized statements defined in methods `newSale()` and `rankImprove()`. The code at ❷ shows `newSale()` is no longer a synchronized method. At ❸, a thread that executes `newSale()` must acquire a lock on the object `objSale` before it can execute the synchronized statements that start from the code at ❸ and end at ❹. Method `newSale()` manipulates variable `copiesSold`. The code at ❺ defines an instance method that includes a synchronized statement that manipulates the instance variable `rank`.

While a thread is executing method `newSale()` on a `Book` instance, another thread can execute method `rankImprove()` on the *same* `Book` instance because `newSale()` and `rankImprove()` acquire locks on monitors of separate objects—that is, `objSale` and `objPos`.

EXAM TIP Multiple threads *can* concurrently execute methods with synchronized statements if they acquire locks on monitors of separate objects.

A thread releases the lock on the object monitor once it exits the synchronized statement block due to successful completion or an exception. If you're trying to modify your shared data using synchronized statements, ensure that the data items are mutually exclusive. As shown in the preceding example, the object references `objSale` and `objPos` refer to different objects.

10.3.4 Immutable objects are thread safe

Immutable objects like an instance of class `String` and all the wrapper classes (like `Boolean`, `Long`, `Integer`, etc.) are thread safe because their contents can't be modified. So no matter how and when you access them, it doesn't result in a *dirty read* or *dirty write*. The exam might ask you to identify a class whose instances are safe from concurrent access and modification, without thread synchronization. The simple answer is the use of immutable objects. In the following example, once `title` is initialized, no matter how it's accessed by multiple threads, it never leads to inconsistent memory or race conditions:

```
class Book{
    private String title;
}
```

EXAM TIP Shared immutable objects can't result in inconsistent object data or an incorrect object state because, once initialized, they can't be modified.

Here's an example of how to create an immutable class:

```
import java.util.Date;
final class BirthDate {
    private final Date birth;
    public BirthDate(Date dob) {
        birth = dob;
    }
    public Date getBirthDate() {
        return (Date)birth.clone();
    }
    public boolean isOlder(Date other) {
        // calculate 'other' with 'birth'
        // return true is 'birth' < 'other'
        return true;
    }
}
```

❶ **final class that can't be extended**

private instance variables ❷ — **Value to instance variables can be assigned only during their instantiation.**

Returns a clone of birth. ❸

❹ **No methods allow modifications to private instance variables of BirthDate.**

The code at ❶ defines class `BirthDate` as a `final` class to ensure that no class can extend the immutable class, and to define additional methods to modify the value of its instance variables. The variable `birth` is defined as a `final` and `private` member at ❷. A private variable can't be manipulated outside the class. Declaring a variable as `final` prevents it from being reassigned a value. But it doesn't prevent its existing value from being modified (remember you can't reassign a value to a `final` reference variable,

but you can modify its value if its methods allow you to). At ❸, `getBirthDate()` returns a clone of `birth`, so that any modifications to the returned value don't change the value of instance variable `birth`. At ❹, you notice that none of the methods of class `BirthDate` allow modification of its instance value `birth`.

In the next "Twist in the Tale" exercise, let's change the positions of a few keywords in class `BirthDate`, defined in the preceding example code, and see whether it will still survive concurrent access by multiple threads and store and report correct data.

Twist in the Tale 10.4

Examine the following class, assuming that java.util.Date is an immutable class. Do you think it will be safe to use it in a multithreaded environment, without any external synchronization of methods that call this class's methods?

```
import java.util.Date;
final class BirthDate {
    private final Date birth;
    public BirthDate(Date dob) {
        birth = dob;
    }
    final public Date getBirthDate() {
        return birth;
    }
    final public boolean isOlder(Date other) {
        // code to compare dates
        return true;
    }
}
```

a Yes

b No

Threads might read and write shared values in their own cache memory. Let's see how you can prevent threads from doing so.

10.3.5 Volatile variables

Apart from using synchronized methods and code blocks, you can use volatile variables to synchronize access to data. Though simpler to use than synchronized code, volatile variables offer only a subset of the features offered by the synchronized code. A synchronized code block can be executed by only one thread, as an atomic operation. Also before a thread gives up the monitor lock on an object, the changes it made to the data become visible to the next thread that acquires the object's monitor lock. The volatile variables don't support the atomicity feature.

A read or write operation on a volatile variable creates a *happens-before* relationship with operations on the same object by other threads. As a result, the compiler can't include optimizations that might enable threads to read the values of volatile variables

from their own local cache. When a thread reads from or writes to a variable (both primitive and reference variables) marked with the keyword `volatile`, it accesses it from the main memory as opposed to storing its copy in the thread's cache memory. This prevents multiple threads from storing a local copy of shared values that might not be consistent across threads.

So if they don't offer features as good as synchronized code, why do we need them? Because they offer simplicity and better performance when defining thread-safe data within certain limits. The modified value of the volatile variables shouldn't be dependent on their current value (like a counter). Also they shouldn't be dependent on other variables for their values. Here's an example of code that defines and uses the volatile variable `closeAcct`:

```
class BankAccount {
    volatile boolean closeAcct = false;
    public void markClosure() {
        closeAcct = true;
    }
    public void closeAccount() {
        if (closeAcct) {
            //..procceed with account closure
        }
    }
}
```

volatile variable closeAcct

New value of closeAcct isn't dependent on its existing value

Another example of using volatile variables is for initializing objects. But if the object is expected to change after initialization, you might need to use additional synchronization to ensure thread-safe data:

```
class StockData{}
class StockExchange{
    static volatile StockData data = null;
    static void loadStockData() {
        // load and initialize reference variable 'data'
        // the variable 'data' is written to only once
    }
}
class BuyStocks {
    public void buy() {
        if (StockExchange.data != null) {
            // analyze data
            // buy stocks
        }
    }
}
```

In this example, if the reference variable data isn't defined as a volatile variable, other classes might access it before it's completely initialized.

Every problem has a solution and every solution has a problem. Though creating multiple threads results in better use of the underlying processor, it can lead to other issues like inconsistent memory, thread deadlock, and more. Let's cover these in detail in the next section.

10.4 Identify and fix code in a multithreaded environment

[10.4] Identify code that may not execute correctly in a multithreaded environment

Threading issues arise when multiple threads work with shared data and when they're dependent on other threads. This section identifies the data that you need to be careful with when working with multiple threads, the operations (or methods) that can lead to threading issues, the order of operations, and how to get around all of these issues.

Let's get started with identifying the data that the threads can share among themselves. Only when you know *what* data can be shared, will you be able to identify *how* it can be fixed.

10.4.1 Variables you should care about

Threads can share the heap that includes class variables and instance variables. Each thread gets its own share of stack memory, which includes local variables, method parameters, and exception handler parameters. So a multithreaded application should safeguard the static and instance variables or attributes of its shared objects. Figure 10.11 shows the type of variables that are always safe in a multithreaded application. So it's always the not-so-safe instance and static variables that you need to care about.

NOTE The terms *static variables* and *static attributes*, and *instance variables* and *instance attributes*, are the same and are often used interchangeably.

But what kind of operations should you care about? Should you only be concerned with the methods that change the value of a shared variable? Or, should you also synchronize the read operations? Let's answer these questions in the next section.

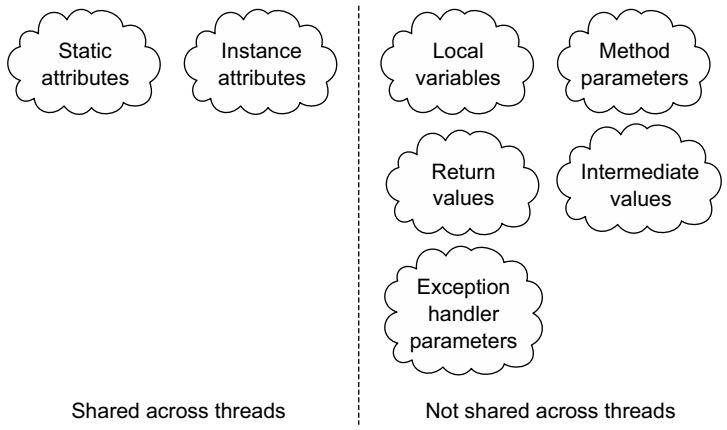

Figure 10.11 Local variables, method parameters, and exception handler parameters are always safe in a multithreaded application.

10.4.2 Operations you should care about

To safeguard your data, you might think you only need to worry about methods that modify the value of a variable. Think again. Methods that *only* read shared variable values can also return incorrect or inconsistent data.

Imagine that agents Shreya and Harry (two threads) manage renting an exhibition ground, say, Axiom (shared resource). Harry has agreed to rent it to a customer and is in the process of signing the legal papers (one thread is updating the shared resource). Shreya has no clue about this development. Just before Harry completes signing the rental agreement, Shreya receives an enquiry about the availability of Axiom and she confirms that it's available (another thread reads data while the shared resource is being modified). In this case, Shreya accessed *inconsistent data*.

Similarly, threads that access shared objects can report *inconsistent memory*. Referring back to the previous example of class `Book` (with an instance variable `copiesSold`), imagine two threads are accessing the same `Book` instance. One thread updates the count of `copiesSold` and the other thread retrieves the value of `copiesSold`. If the second thread retrieves the value of `copiesSold` just before the first thread completes the modification, the second thread returns a *dirty value* that is inconsistent. For instance

```
class Book{
    private int copiesSold = 0;
    public void newSale() {
        ++copiesSold;
    }
    public int getCopiesSold() {
        return copiesSold;
    }
}
class OnlineBuy extends Thread{
    private Book book;
    public OnlineBuy(Book book) {
        this.book = book;
    }
    public void run() {
        book.newSale();
    }
}
class OnlineEnquiry extends Thread{
    private Book book;
    public OnlineEnquiry(Book book) {
        this.book = book;
    }
    public void run() {
        System.out.println(book.getCopiesSold());
    }
}
class InconsistentMemory {
    public static void main(String args[]) throws Exception {
        Book eBook = new Book();
        Thread buy = new OnlineBuy(eBook);
        Thread enquire = new OnlineEnquiry(eBook);
```

When started, OnlineBuy executes newSale() on its instance book.

When started, OnlineEnquiry executes getCopiesSold() on its instance book.

Pass same Book instance book to threads buy and enquire.

```
        buy.start();
        enquire.start();
    }
}
```

Thread enquire might execute ebook.getCopiesSold before thread buy completes execution of eBook.newSale.

In the preceding example, the thread `enquire` might read inconsistent data if it happens to execute `eBook.getCopiesSold()` before the thread `buy` executes `eBook.newSale()`. Depending on how the threads are scheduled, you might see either of the outputs shown in figure 10.12.

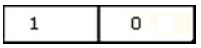

Figure 10.12 Probable outputs of class `InconsistentMemory`

Imagine a thread that produces data and another thread that consumes it. What happens if the producer thread lags behind, leaving the consumer thread without any data to process? Let's work with this situation in the next section.

10.4.3 Waiting for notification of events: using wait, notify, and notifyAll

If interdependent threads share data that's processed alternately by them, the threads should be able to communicate with each other to notify when there's a completion of events, or else the threads might not work correctly.

Imagine you're waiting for your friend to go river rafting at a camp. You check whether she has arrived by peeping out of your tent every minute. This can become quite inconvenient. How about asking her to notify you when she arrives? Let's implement this in code using the `wait` and `notify` methods from class `Object`. Because these methods are defined in class `Object`, they can be called on *any* Java object. Methods `wait()`, `notify()`, and `notifyAll()` enable threads sharing the same data to communicate with each other.

EXAM TIP Methods `wait()`, `notify()`, and `notifyAll()` are defined in class `Object` and not in class `Thread`.

To call `wait()` or `notify()` a thread must own the object's monitor lock. So calls to these methods should be placed within synchronized methods or blocks. Here's an implementation of the previous example in code:

```
class GoRafting extends Thread {
    Friend friend;
    GoRafting(Friend friend) {
        this.friend = friend;
    }
    public void run() {
        System.out.println("Friend reached:" + friend.reached);
        synchronized(friend) {
            try {
                friend.wait();
            }
```

1 Waits for object friend to call either notify() or notifyAll(); call to wait() enclosed within a synchronized block.

```
            catch (InterruptedException e) {
                System.out.println(e);
            }
        }
        System.out.println("Reached:" + friend.reached + ",going rafting");
    }
}
class Friend extends Thread {
    boolean reached = false;

    public void run() {
        while (!reached) {
            try {
                Thread.sleep(2000);
            }
            catch (InterruptedException e) {
                System.out.println(e);
            }
            confirmReached();
        }
    }
    public synchronized void confirmReached() {
        reached = true;
        notify();
    }
}
class Camping {
    public static void main(String args[]) {
        Friend paul = new Friend();
        GoRafting rafting = new GoRafting(paul);
        rafting.start();
        paul.start();
    }
}
```

For this example code, thread sleeps for two seconds.

Calls confirmedReached, which calls notify.

❷ **Call to notify placed in a synchronized method**

❸ **Start thread rafting**

❹ **Start thread friend**

The code at ❸ starts the thread `rafting`. At ❶, when thread `rafting` calls `friend.wait()`, it's placed in the waiting set of threads for the object `friend`, gives up `friend`'s monitor lock, and waits until

- Another thread invokes `notify()` or `notifyAll()` on the same object, `friend`
- Some other thread interrupts `GoRafting`

At ❹, the thread of execution `paul` starts its execution. To execute `notify` at ❷, it must acquire a lock on the monitor for object `paul`. Once it acquires the lock, it executes `notify()`, notifying and waking up one of the threads waiting on `friend`. Because in this example only one thread is waiting on object friend, `notify()` wakes up the thread `rafting`. But waking up the thread `rafting` doesn't guarantee that it will resume its execution immediately. It will enter the `RUNNABLE` (or execution) state and be ready for when the thread scheduler chooses it for execution.

If multiple threads are waiting on an object's monitor, `notify()` wakes up one of these threads. Which thread will be chosen isn't defined and is specific to the JVM implementation. Method `notifyAll()` wakes up all the threads that are waiting on an

object's monitor. These threads compete in the usual manner to acquire a lock on the object's monitor and resume their execution.

When `rafting` resumes its execution, it will send its remaining code to execution.

EXAM TIP Methods `wait()`, `notify()`, and `notifyAll()` *must* be called from a synchronized method or code blocks or else an `IllegalMonitorStateException` will be thrown by the JVM.

The overloaded `wait()` methods enable you to specify a wait timeout:

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

When a thread calls the method `wait()` on an object specifying a timeout duration, it waits until another thread calls `notify()` or `notifyAll` on the same object, it's interrupted by another thread, or the waiting timeout elapses. Even though the overloaded version `wait(long, int)` accepts a timeout duration in nanoseconds, supporting such time precision is JVM- and OS-specific.

EXAM TIP All overloaded versions of `wait()` throw a checked `InterruptedException`. Methods `notify()` and `notifyAll()` don't throw an `InterruptedException`.

Unlike the `Thread`'s method `join()`, which waits for another thread to complete its execution, methods `wait()` and `notify()` don't require a thread to complete their execution. The thread that calls `wait()`, waits for another thread to notify it using method `notify()` or `notifyAll()`. Some other examples from daily life of using `wait()` and `notify()` are teachers and students waiting to be notified for a next workshop on life skills (which can happen multiple times), an operator who answers a phone whenever she's notified of an incoming call, or customers waiting for notification of new offers on a range of televisions.

The exam is sure to question you on the status of a thread when it executes methods from class `Thread` (`sleep()`, `join()`, `yield()`) and methods from class `Object` (`wait()`, `notify()` and `notifyAll()`), and whether they release or retain the object's monitor locks (if they have acquired one). Figure 10.13 summarizes this information.

Method name	Releases lock?	Change in status of executing thread
`sleep()`	✗	`RUNNABLE` → `TIMED_WAITING`
`yield()`	✗	`RUNNABLE` → No change
`join()`	✗	`RUNNABLE` → `WAITING` or `TIMED_WAITING`
`notify()/notifyAll()`	✗	`RUNNABLE` → No change
`wait()`	✓	`RUNNABLE` → `WAITING` or `TIMED_WAITING`

Figure 10.13 Change in thread's status when methods are executed from class `Thread` (`sleep()`, `join()`, `yield()`) and class `Object` (`wait()`, `notify()`, `notifyAll()`)

Multiple threads might deadlock when they have acquired a lock on objects and are waiting to acquire locks on additional objects that are owned by other waiting threads. Let's discuss this threading issue in detail next.

10.4.4 Deadlock

Imagine a tester and a developer are working on two applications, an Android application and an iPhone application. The Android application is in its testing phase; it should be tested and the reported bugs should be fixed by the developer. The iPhone application needs to be developed from scratch; it should be coded by the developer and then tested for bugs. Imagine the tester starts testing the Android application and the developer starts working with the iPhone application. The Android application completes the testing phase and requests for the developer to fix the bugs. At the same time, the iPhone application completes its development phase and requests the tester to test it. The managers working with both the Android and iPhone applications refuse to release their resources (developer or tester) before their project completes. So both projects are waiting for the other project to complete—that is, waiting for the same set of resources. This causes a deadlock—these threads might wait forever. Here's how this looks in code:

```
class Developer {
    synchronized void fixBugs() {
        System.out.println("fixing..");
    }
    synchronized void code() {
        System.out.println("coding..");
    }
}
class Tester {
    synchronized void testAppln() {
        System.out.println("testing..");
    }
}
class AndroidApp extends Thread {
    Developer dev;
    Tester tester;
    AndroidApp(Developer dev, Tester t) {
        this.dev = dev;
        this.tester = t;
    }
    public void run() {
        synchronized(tester) {
            tester.testAppln();
            dev.fixBugs();
        }
    }
}
class iPhoneApp extends Thread {
    Developer dev;
    Tester tester;
```

1 Synchronized methods

2 Acquire lock on tester, reentrant lock on tester, lock on dev

```
    iPhoneApp(Developer dev, Tester t) {
        this.dev = dev;
        this.tester = t;
    }
    public void run() {
        synchronized(dev) {
            dev.code();
            tester.testAppln();
        }
    }
}
class DeadLock {
    public static void main(String args[]) {
        Tester paul = new Tester();
        Developer selvan = new Developer();

        AndroidApp androidApp = new AndroidApp(selvan, paul);
        iPhoneApp iPhoneApp = new iPhoneApp(selvan, paul);

        androidApp.start();
        iPhoneApp.start();
    }
}
```

❸ **Acquire lock on dev, reentrant lock on dev, lock on tester**

❹ **Same developer and tester work with Android and iPhone applications**

The code at ❶ defines synchronized code in classes `Tester` and `Developer`. A thread must acquire a lock on an object's monitor before it can execute its synchronized methods. At ❹, code passes the same `Developer` and `Tester` instances—that is, `paul` and `selvan`—to `AndroidApp` and `iPhoneApp` threads. At ❷, the thread `AndroidApp` first acquires a lock on its instance member `tester`, then reacquires the lock to execute its `testAppln`. It then tries to acquire a lock on its instance member `dev` so that it can call its method `fixBugs()`. But by this time, the code at ❸ has begun its execution, where the thread `iPhoneApp` has already acquired a lock on a shared object `dev` and is waiting to acquire a lock on `tester` so that it can execute `testAppln`. At this point, both the threads `AndroidApp` and `iPhoneApp` deadlock. On the exam, you should be able to recognize the conditions that might lead to thread deadlocking.

10.4.5 Starvation

Imagine you're invited to dine with the president or your favorite sports star, or say a natural calamity strikes. Will it change your existing engagements for the evening? There's a very high probability that it would. Similarly, all application and OS threads that execute on a system are assigned a priority (default or explicit). Usually threads with a higher priority are preferred to execute by the thread scheduler. But this preference might leave threads with a lower priority *starved* to be scheduled.

A thread can also starve to be scheduled when it's waiting to acquire a lock on an object monitor that has been acquired by another thread that usually takes long to execute and is invoked frequently.

Thread scheduling is dependent on an underlying OS. Usually all OSs support prioritized scheduling of high-priority threads, but this behavior isn't guaranteed across different platforms.

10.4.6 Livelock

Imagine you're talking with your friend on your mobile phone and the call drops. You try to reconnect with her, but her phone is busy (because she's trying to call you!). You wait for a few moments and try to reconnect, only to discover that her phone is still busy, because she waited for exactly the same duration before trying to reconnect again. If you compare yourself and your friend with threads, you both are in a livelock. You and your friend aren't blocked—both of you are responding but aren't being able to do what you both intend to (talk with each other).

EXAM TIP Threads in a livelock aren't blocked; they're responding to each other, but they aren't able to move to completion.

Because a livelock depends on the exact time of execution of threads, the same set of threads might not always livelock.

10.4.7 Happens-before relationship

With threads, there's little that can be guaranteed. You can't determine exactly when a particular method will start, pause, or resume its execution. You can't guarantee the sequence in which multiple threads might execute. On the exam, you'll be questioned on the correct, incorrect, or probable outcome of code that defines or works with threads. The Java language uses a *happens-before* relationship, which is when one task is guaranteed to happen before another in a multithreading environment. Figure 10.14 shows this relationship.

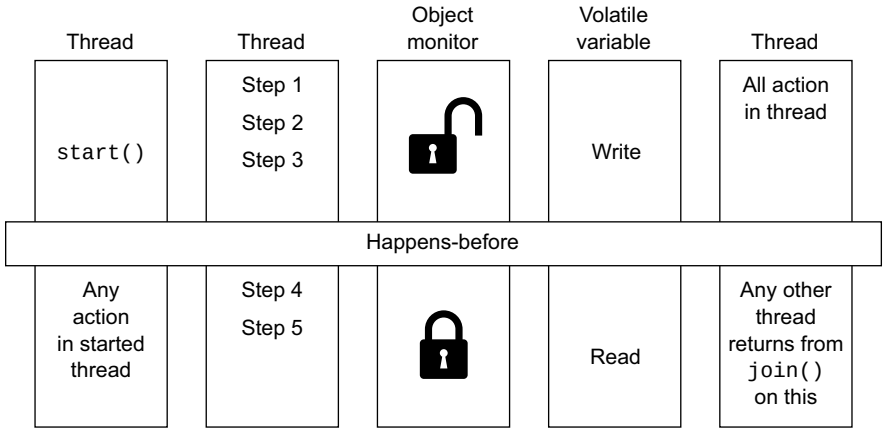

Figure 10.14 Happens-before relationship

The *happens-before* relationship, as included in Java's language specification, will enable you to answer a lot of questions on threading on the exam. For example

- The execution of `start()` *happens-before* any action in a thread is started.
- When code is defined in a sequence, step 1 *happens-before* step 2.
- Unlocking of an object monitor *happens-before* any other thread acquires a lock on it.
- A write to a volatile field *happens-before* every subsequent read of that field.
- All actions in a thread *happens-before* any other thread returns from a join on that thread.

As you attempt the sample exam questions at the end of this chapter, mark the answers that could be determined using the happens-before relationship. If you discover other threading behavior that can be added to the happens-before relationship, please notify me too!

10.5 Summary

This chapter covers how threads enable systems to use their single or multiple processors to execute applications concurrently. We worked with class `Thread` and interface `Runnable` to define and create threads. Threads can exist in multiple states (`NEW`, `RUNNABLE`, `WAITING`, `TIMED_WAITING`, `BLOCKED` and `TERMINATED`), and we used multiple methods (`start()`, `yield()`, `join()`, `sleep()`, `wait()`, `notify()`, `notifyAll()`) to transition threads from one state to another. With multithreading you can guarantee very little when it comes to commanding or determining when a thread actually transitions from one state to another. The actual thread scheduling is handled by the underlying OS.

We covered how threads can be independent or dependent. Dependent threads communicate with each other by sharing data, which can lead to thread interference and memory inconsistency issues. Prevention techniques include synchronizing thread access to data by acquiring locks and defining atomic operations. We used synchronized methods and code blocks to synchronize data access. The volatile variables, though not as powerful and effective as synchronized methods, offer a simple solution to safeguard data from multiple concurrent thread access.

Using locks on objects safeguards object data, but it also leads to thread contention. Other threading issues include deadlock, starvations, and livelock.

REVIEW NOTES

This section lists the main points covered in this chapter.

Create and use threads

- All nontrivial Java applications are multithreaded.
- Multiple threads can be supported by an underlying system by using multiple hardware processors, by time-slicing a single processor, or by time-slicing multiple processors.

- Implementation of Java threads is JVM-specific.
- Though related, a `Thread` instance and a thread of execution aren't the same. A `Thread` instance is a Java object.
- The main thread is named `main` by the JVM. Don't confuse it with the method `main()`.
- Class `Thread` and interface `Runnable` can be used to create and start a new thread of execution.
- To create your own thread objects using class `Thread`, you must extend it and override its method `run()`.
- When you call `start()` on a `Thread` instance, it creates a new thread of execution.
- When a new thread of execution starts, it will execute the code defined in the thread instance's method `run()`. Method `start()` will trigger the creation of a new thread of execution, allocating resources to it.
- Because you can't be sure of the order of execution of threads by an underlying OS, multithreaded code might output different results when executed on the same or a different system.
- When you create a thread class by extending class `Thread`, you lose the flexibility of inheriting any other class.
- When you implement the `Runnable` interface, you must implement its method `run()`.
- If your class implements the `Runnable` interface, then you should pass its instance to the constructor of class `Thread`.
- The `Thread` constructor accepts a `Runnable` object. A `Thread` instance stores a reference to a `Runnable` object and uses it when you start its execution (by calling `start()`).
- Because class `Thread` implements the `Runnable` interface, you can instantiate a thread by passing it another `Thread` instance.
- Each thread is created with a priority. Its range varies from 1 to 10, with 1 being the lowest priority and 10 the highest priority. By default, a thread creates another thread with the same priority as its own.
- You can't guarantee that a thread with a higher priority will always execute before a thread with a lower priority.

Thread lifecycle

- You can control the transition of a thread from one state to another by calling its methods.
- The exact time of thread state transition is controlled by a thread scheduler, which is bound to vary across platforms.
- A thread can exist in multiple states: `NEW`, `RUNNABLE`, `WAIT`, `TIMED_WAITING`, `BLOCKED`, or `TERMINATED`.

- A thread that hasn't yet started is in the `NEW` state.
- Calling `start()` on a new thread instance implicitly calls its method `run()`, which transitions its state from `NEW` to READY.
- A thread in the `RUNNABLE` state is all set to be executed. It's just waiting to be chosen by the thread scheduler so that it gets the processor time.
- As a programmer, you can't control or determine when a particular thread transitions from the READY state to the RUNNING state, and when it actually gets to execute.
- The states READY and RUNNING are together referred to as the `RUNNABLE` state.
- A thread in the `RUNNABLE` state is executing in the JVM, but it may be waiting for other resources from the OS, such as a processor.
- A thread that's blocked waiting for a monitor lock is in the `BLOCKED` state.
- A thread that's waiting for another thread to perform an action for up to a specified waiting time is in the `TIMED_WAITING` state.
- A RUNNING thread enters the `TIMED_WAITING` state when you call `sleep(int)`, `join(int)`, or `wait(int)` on it.
- A thread that's waiting indefinitely for another thread to perform a particular action is in the `WAITING` state.
- When you call `wait()` on a RUNNING thread, it transitions to the `WAITING` state. It can change back to the READY state when `notify()` or `notifyAll()` is called.
- A RUNNING thread might enter the `BLOCKED` state when it's waiting for other system resources like network connections or to acquire an object lock to execute a synchronized method or code block. Depending on whether the thread is able to acquire the monitor lock or resources, it returns back to the READY state.
- With the successful completion of `run()`, a thread is in the `TERMINATED` state.
- A thread might transition from any state to the `TERMINATED` state due to an exception.

Methods of class Thread

- Calling `start()` on a `Thread` instance creates a new thread of execution, which executes `run()`.
- You can call `start()` on a thread that's in the `NEW` state. Calling `start()` from any other thread state will throw an `IllegalThreadStateException`.
- Calling `run()` on a `Thread` instance doesn't start a new thread of execution. The `run()` continues to execute in the same thread.
- A thread might pause its execution due to the calling of an explicit method or when its time slice with the processor expires.
- Method `yield()` makes the currently executing thread pause its execution and give up its current use of the processor. But it only acts as a hint to the scheduler. The scheduler might also ignore it.

- Method `yield()`is static. It can be called from any method, and it doesn't throw any exceptions.
- Method `yield()` can be placed literally anywhere in your code—not only in method `run()`.
- Method `sleep()` is guaranteed to cause the currently executing thread to temporarily give up its execution for at least the specified number of milliseconds (and nanoseconds).
- Unless interrupted, the currently executing thread will sleep for at least the specified duration. It might not start its execution immediately after the specified time elapses.
- Method `sleep()` is `Thread`'s static method and it makes the currently executing thread give up its execution. Because all code is executed by some thread, placement of `sleep()` will determine which `Thread` instance will give up its execution.
- A thread that's suspended due to a call to `sleep()` doesn't lose ownership of any monitors.
- If thread `A` calls `join()` on a `Thread` instance `B`, `A` will wait for `B` to complete its execution before `A` can proceed to its own completion.
- Method `join()` guarantees that the calling thread won't execute its remaining code until the thread on which it calls `join()` completes.
- A thread can pause its execution and wait on an object, a queue in this case, by calling `wait()`, until another thread calls `notify()` or `notifyAll()` on the same object.
- Methods `wait()`, `notify()`, and `notifyAll()` can be called on all Java objects, because they're defined in class `Object` and not class `Thread`.
- A thread completes its execution when its method `run()` completes.

Protect shared data

- Interleaving of multiple threads that manipulate shared data using multiple steps leads to thread interference.
- A simple statement like incrementing a variable value might involve multiple steps like loading of the variable value from memory to registers (working space), incrementing the value, and reloading the new value in memory.
- When multiple threads execute this seemingly atomic statement, they might interleave, resulting in incorrect variable values.
- Making your applications thread safe means securing your shared data so that it stores correct data, even when it's accessed by multiple threads.
- Thread safety isn't about safe threads—it's about safeguarding your shared data that might be accessible to multiple threads.
- A thread-safe class stores correct data without requiring calling classes to guard it.
- You can lock objects by defining synchronized methods and synchronized statements.

- Synchronized methods are defined by prefixing the definition of a method with the keyword `synchronized`. You can define both instance and static methods as synchronized methods.
- For nonstatic synchronized methods, a thread locks the monitor of the object on which the synchronized method is called. To execute static synchronized methods, a thread locks the monitor associated with the `Class` object of its class.
- A thread releases the lock on an object monitor after it exits a synchronized method, whether due to successful completion or due to an exception.
- To execute synchronized statements, a thread must acquire a lock on an object monitor. For instance methods an implicit lock is acquired on the object on which it's called. For synchronized statements, you can specify an object to acquire a lock on.
- To execute synchronized statements, a lock must be acquired before the execution of the statements.
- Multiple threads can concurrently execute methods with synchronized statements if they acquire a lock on monitors of separate objects.
- A thread releases the lock on the object monitor once it exits the synchronized statement block due to successful completion or an exception.
- Immutable objects like an instance of class `String` and the wrapper classes (like `Boolean`, `Long`, `Integer`, etc.) are thread safe because their contents can't be modified.
- You can define an immutable class by limiting access to its attributes within the class and not defining any methods to modify its state.
- Once initialized, an immutable instance doesn't allow modification to its value.
- You can use volatile variables to synchronize access to data.
- When a thread reads from or writes to a variable (both primitive and reference variables) marked with the keyword `volatile`, it accesses it from the main memory, as opposed to storing its copy in the thread's cache memory. This prevents multiple threads from storing a local copy of shared values that might not be consistent across threads.

Identify and fix code in a multithreaded environment

- Threading issues arise when multiple threads work with shared data and when they're dependent on other threads.
- Local variables, method parameters, and exception handler parameters are always safe in a multithreaded application.
- Class and instance variables might not always be safe in a multithreaded application.
- Methods `wait()`, `notify()`, and `notifyAll()` can be used for interthread notification.

- To call `wait()` or `notify()` a thread must own the object's monitor lock. So calls to these methods should be placed within synchronized methods or blocks or else an `IllegalMonitorStateException` will be thrown by the JVM.
- All overloaded versions of `wait()` throw a checked `InterruptedException`. Methods `notify()` and `notifyAll()` don't throw an `InterruptedException`.
- Unlike `Thread`'s method `join()`, which waits for another thread to complete its execution, methods `wait()` and `notify()` don't require a thread to complete their execution.
- Multiple threads might deadlock when they have acquired a lock on objects and are waiting to acquire locks on additional objects that are owned by other waiting threads.
- All threads are assigned a priority, either implicitly or explicitly. Usually threads with a higher priority are preferred to execute by the thread scheduler. But this preference might leave threads with a lower priority starved to be scheduled.
- A thread can also starve to be scheduled when it's waiting to acquire a lock on an object monitor that has been acquired by another thread that usually takes long to execute and is invoked frequently.
- Threads in a livelock aren't blocked; they're responding to each other, but they aren't able to move to completion.
- With threads, there's little that can be guaranteed. The Java language uses a happens-before relationship, which is when one task is guaranteed to happen before another in a multithreading environment.
- The execution of `start()` happens-before any action in a thread is started.
- When code is defined in a sequence, step 1 happens-before step 2.
- Unlocking of an object monitor happens-before any other thread acquires a lock on it.
- A write to a volatile field happens-before every subsequent read of that field.
- All actions in a thread happens-before any other thread returns from a join on that thread.

SAMPLE EXAM QUESTIONS

Q 10-1. What is the probable output of the following code?

```
enum Seasons{SPRING,SUMMER}
class ETree extends Thread {
    String name;
    public ETree(String name) {this.name = name;}
    public void run() {
        for (Seasons season : Seasons.values())
            System.out.print(name + "-" + season + " ");
    }
    public static void main(String args[]) {
        ETree oak = new ETree("Oak"); oak.start();
```

```
        ETree maple = new ETree("Maple"); maple.start();
    }
}
```

a `Oak-SPRING Maple-SPRING Oak-SUMMER Maple-SUMMER`
b `Oak-SUMMER Oak-SPRING Maple-SPRING Maple-SUMMER`
c `Oak-SUMMER Maple-SUMMER Oak-SPRING Maple-SPRING`
d `Oak-SPRING Oak-SUMMER Maple-SPRING Maple-SUMMER`
e `Maple-SPRING Maple-SUMMER Oak-SPRING Oak-SUMMER`
f `Maple-SPRING Oak-SPRING Oak-SUMMER Maple-SUMMER`
g Compilation error
h Runtime exception

Q 10-2. Examine the following code and select the correct options.

```
public class EThread {
    public static void main(String[] args) {
        Thread bug = new Thread() {
            public void run() {
                System.out.print("check bugs");
            }
        };
        Thread reportQA = new Thread(bug);
        reportQA.run();
    }
}
```

a No code output
b Code prints `check bugs` once.
c Code prints `check bugs` twice.
d Code prints `check bugs` in an infinite loop.
e Replacing `reportQA.run()` with `reportQA.start()` will throw a compilation exception.
f Replacing `reportQA.run()` with `reportQA.start()` will generate the same output on the system's console.

Q 10-3. What is the output of the following code?

```
1. class EPen implements Runnable {
2.     public void run() {
3.         System.out.println("eJava");
4.         start();
5.     }
6.     public static void main(String... args) {
7.         new Thread(new EPen()).start();
8.     }
9. }
```

a. It prints `eJava` once.
b. It prints `eJava` multiple times.
c. Compilation error
d. Runtime exception

Q 10-4. The thread `paul` must start only after the thread `shreya` has completed its execution. Which of the following code options, when inserted at `//INSERT CODE HERE`, will ensure this?

```
1. class EJob extends Thread {
2.     public void run() {
3.         System.out.println("executing");
4.     }
5.     public static void main(String[] args) {
6.        Thread paul = new EJob();
7.        Thread shreya = new EJob();
8.        shreya.start();
9.        paul.start();
10.        //INSERT CODE HERE
11.    }
12. }
```

a. `shreya.join();`
b. `paul.join();`
c. `shreya.sleep(1000);`
d. `shreya.wait();`
e. `paul.notify();`
f. None of the above

Q 10-5. What is the output of the following code?

```
class ELock{}
class EPaper implements Runnable {
    public void run() {
        synchronized(ELock.class) {
            System.out.println("Hand made paper");
        }
    }
    public static void main(String args[]) throws Exception {
        Thread epaper = new Thread(new EPaper());
        epaper.start();
        synchronized(ELock.class) {
            epaper.join();
        }
    }
}
```

a. The threads `main` and `epaper` will always deadlock.
b. The threads `main` and `epaper` might not deadlock.

c Compilation error
d Runtime exception

Q 10-6. Selvan is testing a multithreaded application in which thread A downloads data in a hash map. Thread B uses the data from the same hash map and displays it to a user for modification. Thread C is supposed to save the modified data and replace the existing data in the hash map. When a user tries to save the data, the application stops responding. What could be the probable reasons?

a Thread A and thread C deadlock.
b Thread B and thread C deadlock.
c Thread C is discovered to be a high-priority thread that performs complex calculations and doesn't allow other threads to execute.
d Thread C throws an exception.

Q 10-7. Instances of which of the following classes will always be safe to use in a multithreaded environment?

a
```
class ESafe {final int value; /*constructor to initialize value*/}
```
b
```
class ESafe {final Object value; /*constructor to initialize value*/}
```
c
```
final class ESafe {
      ESafe(Object obj) {value = obj;}
      private final Object value;
      synchronized Object read() {return value;}
      synchronized void modify(Object obj) {}
  }
```
d
```
class ESafe {final String value; /*constructor to initialize value*/}
```

Q 10-8. What is the output of the following code?

```
class EProcess extends Thread {
    public void run() {
        this.yield();                       //line1
        for (int i = 0; i < 1000; i++)
            System.out.print(i);
    }
    public static void main(String... args) {
        Thread myThread = new EProcess();
        myThread.run();                     //line2
    }
}
```

a On execution, code at line 1 might make the thread `main` give up its execution slot.
b On execution, code at line 1 makes the thread `myThread` give up its execution slot.
c Compilation error at line 1
d Runtime exception at line 2

Q 10-9. Select the correct options for the following code.

```
class EAppln extends Thread {
    String name;
    EAppln(String name) { this.name = name; }
    public void run() throws InterruptedException {     //1
        sleep(1000);                                    //2
        System.out.println("executing-" + name);
    }
}
class EJava {
    public static void main(String args[]) {
        EAppln app1 = new EAppln("Detect");
        EAppln app2 = new EAppln("Analyze");
        app1.start();
        app2.start();
    }
}
```

a The output is

```
executing-Detect
executing-Analyze
```

b The output is

```
executing-Analyze
executing-Detect
```

c The thread `appl` sleeps for a maximum of 1000 milliseconds before it's rescheduled to run again.

d The treads `app1` and `app2` might sleep at the same time.

e Compilation error

f Runtime exception

Q 10-10. On execution, class `DeclareResults` might not display all of the four string values passed to `ProcessData` constructors at lines 18 and 19. How can you fix this situation?

```
1.  import java.util.*;
2.  class Admission{
3.     static int id;
4.     static Map<Integer, String> results = new HashMap<>();
5.     static int getNextId() {return ++id; }
6.     static void qualify(Integer i, String s) {results.put(i, s);}
7.  }
8.  class ProcessData extends Thread{
9.     List<String> list;
10.    ProcessData(String[] val) {list = Arrays.asList(val);}
11.    public void run() {
12.        for (String item : list)
13.            Admission.qualify(Admission.getNextId(), item);
14.    }
15. }
16. class DeclareResults {
17.    public static void main(String args[]) throws Exception {
```

```
18.         ProcessData thread1 = new ProcessData(
                                        new String[]{"Paul", "Shreya"});
19.         ProcessData thread2 = new ProcessData(
                                        new String[]{"Shreya", "Harry"});
20.         thread1.start(); thread2.start();
21.         thread1.join(); thread2.join();
22.         for (String name : Admission.results.values())
23.             System.out.println(name);
24.     }
25. }
```

a. By declaring only the `getNextId` method as synchronized at line 5
b. By declaring only the `qualify` method as synchronized at line 6
c. By declaring both the `getNextId` and `qualify` methods as synchronized at lines 5 and 6
d. By passing unique values to the threads `thread1` and `thread2` at lines 18 and 19
e. By changing the type of variable `results` from `Map` to `List`
f. None of the above

ANSWERS TO SAMPLE EXAM QUESTIONS

A 10-1. a, d, e, f

[10.1] Create and use the Thread class and the Runnable interface

Explanation: Each thread instance `oak` and `maple`, when started, will output the values of enum `Seasons`—that is, `SPRING` and `SUMMER` (always in this order). The order of the elements returned by `Seasons.values()` isn't random. The enum values are always returned in the order in which they are defined.

You can't guarantee whether thread `oak` completes or begins its execution before or after thread `maple`. The thread scheduler can start `oak`, make it print `Oak-SPRING`, run `maple` so that it prints `Maple-SPRING`, return the control to `oak`, or run `maple` to completion. Whatever the sequence, the happens-before contract guarantees that code in a thread executes in the order it's defined. So the thread `oak` or `maple` can never print the enum value `SUMMER` before the enum value `SPRING`.

A 10-2. b, f

[10.1] Create and use the Thread class and the Runnable interface

Explanation: The following code creates an anonymous class that subclasses class `Thread`. Its instance is referred by `bug`, a reference variable of type `Thread`.

```
Thread bug = new Thread() {
    public void run() {
        System.out.print("check bugs");
    }
};
```

Because class `Thread` implements the `Runnable` interface, you can pass its instance as a target object to instantiate another `Thread` instance, `reportQA`:

```
Thread reportQA = new Thread(bug);
```

The variable `reportQA` refers to an anonymous class instance that overrides its method `run()`. So calling `reportQA.run()` executes the overridden method `run()` and prints `check bugs` only once.

Option (f) is correct. Even though calling `reportQA.run()` doesn't start a separate thread of execution and `reportQA.start()` does, both will print `check bugs` once on the system's console.

A 10-3. c

[10.1] Create and use the Thread class and the Runnable interface

Explanation: Class `EPen` implements the `Runnable` interface; it doesn't extend class `Thread`. So it doesn't have access to method `start()`. Calling `start()` at line 4 results in the compilation failure.

A 10-4. f

[10.2] Manage and control thread lifecycle

Explanation: Calling `join` from a thread ensures that the thread on which `join` is called completes before the calling thread completes its execution. If the thread `main` calls `shreya.join()` before starting the thread `paul`, it can ensure that `paul` will start after `shreya` completes its execution. For this to happen, `main` should call `shreya .join()` at line 9 and `paul.start()` at line 10.

Options (a) and (b) are incorrect. Calling `shreya.join()` or `paul.join()` at line 10 will ensure that these threads complete their execution before the thread `main` completes its own execution.

Option (c) is incorrect. The thread `main` executes `shreya.sleep()`. Because `sleep()` is a static method, it will make `main` sleep for at least the specified time in milliseconds (if not interrupted).

Options (d) and (e) are incorrect because they won't serve the purpose. Method `wait()` makes a thread release its object lock and makes it wait until another object that has acquired a lock on it calls `notify()` or `notifyAll()`. Also `wait()`, `notify()`, and `notifyAll()` must be called from a synchronized block of code or else JVM will throw an `IllegalMonitorStateException`.

A 10-5. b

[10.4] Identify code that may not execute correctly in a multi-threaded environment

Explanation: You can't determine the exact time that a scheduler starts the execution of a thread. So the thread `epaper` can acquire the lock on a monitor associated with

class `ELock`, execute the `epaper`'s method `run()`, and exit *before* method `main()` gets its turn to acquire a lock on `ELock.class` and call `epaper.join()`.

A 10-6. a, b, c

[10.3] Synchronize thread access to shared data

Explanation: The application stops responding after a user tries to save data. So the issue is the series of actions that execute when thread C is initiated. All the three threads, A, B, and C, are sharing the same data set. Thread C might try to write data that is being modified by thread A or B. So thread C might deadlock with thread A or B.

Option (c) is correct. Saving the data is supposed to start thread C. An application might appear to stop responding if it doesn't respond to user events. A scheduler chooses from a pool of threads to execute the threads that are ready for execution. Though OS-specific, threads of high priority are usually preferred over threads with lower priority.

Option (d) is incorrect. An unhandled exception should make the application exit.

A 10-7. a, c, d

[10.4] Identify code that may not execute correctly in a multi-threaded environment

Explanation: Option (a) is correct. Once assigned a value, a primitive variable can't be changed. So it's safe from reading and writing inconsistent or dirty values.

Option (b) is incorrect. Even though defining the instance variable value of class `ESafe` as a `final` member prevents reassigning a value to it, it doesn't prevent modifications to its state. The reference variable `value` isn't private, so it's accessible outside class `ESafe`.

Option (c) is correct. Marked with the keyword `private`, `value` isn't accessible outside class `ESafe`. Also, access to the state of `ESafe` is synchronized and is safe from concurrent reading or writing by multiple threads.

Option (d) is correct. Once initialized, the state of class `ESafe` (value of variable value) can't be modified because class `String` is immutable.

A 10-8. a

[10.2] Manage and control thread lifecycle

Explanation: The code at line 2 doesn't start a new thread of execution. So `myThread.run()` executes in the `main` thread and not a separate thread. When `main` executes `this.yield()` (`yield()` is a static method), it might make `main` give up its execution slot and wait until the scheduler allows it to run again.

A 10-9. e

[10.2] Manage and control thread lifecycle

Explanation: Method `run()` in class `EAppln` can't override method `run()` in class `Thread` by declaring to throw a checked `InterruptedException`. The class fails to compile.

A 10-10. c

[10.3] Synchronize thread access to shared data
[10.4] Identify code that may not execute correctly in a multi-threaded environment

Explanation: The variable `results`, a hash map, stores unique keys. Concurrent access to the unsynchronized `getNextId()` can return the same ID values if it's accessed concurrently by multiple threads. This can lead to overriding values for the same keys in a hash map. So `getNextId()` should be declared as a synchronized method at line 5.

Method `qualify()` must also be declared as a synchronized method to make `Map` results thread safe; concurrent updates to a `Map` can result in overriding values. Internally, a hash map uses hash values of its keys to determine in which bucket it should add a value. When a hash map adds multiple values to a bucket, it links them so that it can retrieve all the values for a bucket value. But concurrent modification of a `Map` object might report an empty bucket to multiple threads and so no linking will happen between the values added to the same bucket. This can lead to untraceable values.