**Section 23.1 Introduction**

• Two tasks that are operating concurrently are both making progress at once.

• Two tasks that are operating in parallel are executing simultaneously. In this sense, parallelism is a subset of concurrency. Today’s multi-core computers have multiple processors that can perform tasks in parallel.

• Java makes concurrency available to you through the language and APIs.

• Java programs can have multiple threads of execution, where each thread has its own methodcall stack and program counter, allowing it to execute concurrently with other threads. This capability is called multithreading.

• In a multithreaded application, threads can be distributed across multiple processors (if available) so that multiple tasks execute in parallel and the application can operate more efficiently.

• The JVM creates threads to run a program and for housekeeping tasks such as garbage collection.

• Multithreading can also increase performance on single-processor systems—when one thread cannot proceed (because, for example, it’s waiting for the result of an I/O operation), another can use the processor.

• The vast majority of programmers should use existing collection classes and interfaces from the concurrency APIs that manage synchronization for you.

#### Section 23.2 Thread States and Life Cycle

• A new thread begins its life cycle in the new state (p. [961](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec5_html#page_961)). When the program starts the thread, it’s placed in the runnable state. A thread in the runnable state is considered to be executing its task.

• A runnable thread transitions to the waiting state (p. [961](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec5_html#page_961)) to wait for another thread to perform a task. A waiting thread transitions to runnable when another thread notifies it to continue executing.

• A runnable thread can enter the timed waiting state (p. [961](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec5_html#page_961)) for a specified interval of time, transitioning back to runnable when that time interval expires or when the event it’s waiting for occurs.

• A runnable thread can transition to the timed waiting state if it provides an optional wait interval when it’s waiting for another thread to perform a task. Such a thread will return to the runnablestate when it’s notified by another thread or when the timed interval expires.

• A sleeping thread (p. [961](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec5_html#page_961)) remains in the timed waiting state for a designated period of time, after which it returns to the runnable state.

• A runnable thread transitions to the blocked state (p. [961](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec5_html#page_961)) when it attempts to perform a task that cannot be completed immediately and the thread must temporarily wait until that task completes. At that point, the blocked thread transitions to the runnable state, so it can resume execution.

• A runnable thread enters the terminated state (p. [961](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec5_html#page_961)) when it successfully completes its task or otherwise terminates (perhaps due to an error).

• At the operating-system level, the runnable state (p. [962](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec10_html#page_962)) encompasses two separate states. When a thread first transitions to the runnable state from the new state, it’s in the ready state (p.[962](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec10_html#page_962)). A ready thread enters the running state (p. [962](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec10_html#page_962)) when the operating system dispatches it.

• Most operating systems allot a quantum (p. [962](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec10_html#page_962)) in which a thread performs its task. When this expires, the thread returns to the ready state and another thread is assigned to the processor.

• Thread scheduling determines which thread to dispatch based on thread priorities.

• The job of an operating system’s thread scheduler (p. [962](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec10_html#page_962)) is to determine which thread runs next.

• When a higher-priority thread enters the ready state, the operating system generally preempts the currently running thread (an operation known as preemptive scheduling; p. [963](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec11_html#page_963)).

• Depending on the operating system, higher-priority threads could postpone—possibly indefinitely (p. [963](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec11_html#page_963))—the execution of lower-priority threads.

#### Section 23.3 Creating and Executing Threads with the Executor Framework

• A Runnable (p. [963](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec11_html#page_963)) object represents a task that can execute concurrently with other tasks.

• Interface Runnable declares method run (p. [963](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec11_html#page_963)) in which you place the code that defines the task to perform. The thread executing a Runnable calls method run to perform the task.

• A program will not terminate until its last thread completes execution.

• You cannot predict the order in which threads will be scheduled, even if you know the order in which they were created and started.

• It’s recommended that you use the Executor interface (p. [963](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec11_html#page_963)) to manage the execution ofRunnable objects. An Executor object typically creates and manages a group of threads—called a thread pool (p. [963](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec11_html#page_963)).

• Executors (p. [964](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec14_html#page_964)) can reuse existing threads and can improve performance by optimizing the number of threads to ensure that the processor stays busy.

• Executor method execute (p. [963](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec11_html#page_963)) receives a Runnable and assigns it to an available thread in a thread pool. If there are none, the Executor creates a new thread or waits for one to become available.

• Interface ExecutorService (of package java.util.concurrent; p. [964](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec14_html#page_964)) extends interfaceExecutor and declares other methods for managing the life cycle of an Executor.

• An object that implements the ExecutorService interface can be created using static methods declared in class Executors (of package java.util.concurrent).

• Executors method newCachedThreadPool (p. [965](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec16_html#page_965)) returns an ExecutorService that creates new threads as they’re needed by the application.

• ExecutorService method execute executes its Runnable sometime in the future. The method returns immediately from each invocation—the program does not wait for each task to finish.

• ExecutorService method shutdown (p. [967](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec18_html#page_967)) notifies the ExecutorService to stop accepting new tasks, but continues executing existing tasks and terminates when those tasks complete execution.

#### Section 23.4 Thread Synchronization

• Thread synchronization (p. [968](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec4_html#page_968)) coordinates access to shared mutable data by multiple concurrent threads.

• By synchronizing threads, you can ensure that each thread accessing a shared object excludes all other threads from doing so simultaneously—this is called mutual exclusion (p. [968](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec4_html#page_968)).

• A common way to perform synchronization is to use Java’s built-in monitors. Every object has a monitor and a monitor lock (p. [968](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec4_html#page_968)). The monitor ensures that its object’s monitor lock is held by a maximum of only one thread at any time, and thus can be used to enforce mutual exclusion.

• If an operation requires the executing thread to hold a lock while the operation is performed, a thread must acquire the lock (p. [968](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec4_html#page_968)) before it can proceed with the operation. Any other threads attempting to perform an operation that requires the same lock will be blocked until the first thread releases the lock, at which point the blocked threads may attempt to acquire the lock.

• To specify that a thread must hold a monitor lock to execute a block of code, the code should be placed in a synchronized statement (p. [968](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec4_html#page_968)). Such code is said to be guarded by the monitor lock.

• The synchronized statements are declared using the synchronized keyword:

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#p1037pro01a)

synchronized (*object*)  
{  
   *statements*  
} // end synchronized statement

where object is the object whose monitor lock will be acquired; object is normally this if it’s the object in which the synchronized statement appears.

• Java also allows synchronized methods (p. [969](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec23_html#page_969)). Before executing, a synchronized instance method must acquire the lock on the object that’s used to call the method. Similary, a static synchronized method must acquire the lock on the class that’s used to call the method.

• ExecutorService method awaitTermination (p. [973](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec24_html#page_973)) forces a program to wait for threads to terminate. It returns control to its caller either when all tasks executing in the ExecutorServicecomplete or when the specified timeout elapses. If all tasks complete before the timeout elapses, the method returns true; otherwise, it returns false.

• You can simulate atomicity (p. [974](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec25_html#page_974)) by ensuring that only one thread performs a set of operations at a time. Atomicity can be achieved with synchronized statements or synchronized methods.

• When you share immutable data across threads, you should declare the corresponding data fieldsfinal to indicate that variables’ values will not change after they’re initialized.

#### Section 23.5 Producer/Consumer Relationship without Synchronization

• In a multithreaded producer/consumer relationship (p. [976](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec25_html#page_976)), a producer thread generates data and places it in a shared object called a buffer. A consumer thread reads data from the buffer.

• Operations on a buffer data shared by a producer and a consumer should proceed only if the buffer is in the correct state. If the buffer is not full, the producer may produce; if the buffer is not empty, the consumer may consume. If the buffer is full when the producer attempts to write into it, the producer must wait until there’s space. If the buffer is empty or the previous value was already read, the consumer must wait for new data to become available.

#### Section 23.6 Producer/Consumer Relationship: ArrayBlockingQueue

• ArrayBlockingQueue (p. [984](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec34_html#page_984)) is a fully implemented buffer class from packagejava.util.concurrent that implements the BlockingQueue interface.

• An ArrayBlockingQueue can implement a shared buffer in a producer/consumer relationship. Method put (p. [984](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec34_html#page_984)) places an element at the end of the BlockingQueue, waiting if the queue is full. Method take (p. [984](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec34_html#page_984)) removes an element from the head of the BlockingQueue, waiting if the queue is empty.

• ArrayBlockingQueue stores shared mutable data in an array that’s sized with an argument passed to the constructor. Once created, an ArrayBlockingQueue is fixed in size.

#### Section 23.7 (Advanced) Producer/Consumer Relationship with synchronized, wait,notify and notifyAll

• You can implement a shared buffer yourself using the synchronized keyword and Objectmethods wait, notify and notifyAll (p. [988](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec7_html#page_988)).

• A thread can call Object method wait to release an object’s monitor lock, and wait in the waitingstate while the other threads try to enter the object’s synchronized statement(s) or method(s).

• When a thread executing a synchronized statement (or method) completes or satisfies the condition on which another thread may be waiting, it can call Object method notify (p. [988](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec7_html#page_988)) to allow a waiting thread to transition to the runnable state. At this point, the thread that was transitioned can attempt to reacquire the monitor lock on the object.

• If a thread calls notifyAll (p. [988](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec7_html#page_988)), then all the threads waiting for the monitor lock become eligible to reacquire the lock (that is, they all transition to the runnable state).

#### Section 23.8 (Advanced) Producer/Consumer Relationship: Bounded Buffers

• You cannot make assumptions about the relative speeds of concurrent threads.

• A bounded buffer (p. [995](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec44_html#page_995)) can be used to minimize the amount of waiting time for threads that share resources and operate at the same average speeds. If the producer temporarily produces values faster than the consumer can consume them, the producer can write additional values into the extra buffer space (if any are available). If the consumer consumes faster than the producer produces new values, the consumer can read additional values (if there are any) from the buffer.

• The key to using a bounded buffer with a producer and consumer that operate at about the same speed is to provide the buffer with enough locations to handle the anticipated “extra” production.

• The simplest way to implement a bounded buffer is to use an ArrayBlockingQueue for the buffer so that all of the synchronization details are handled for you.

#### Section 23.9 (Advanced) Producer/Consumer Relationship: The Lock and ConditionInterfaces

• The Lock and Condition interfaces (p. [1002](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec50_html#page_1002)) give programmers more precise control over thread synchronization, but are more complicated to use.

• Any object can contain a reference to an object that implements the Lock interface (of packagejava.util.concurrent.locks). A thread calls a Lock’s lock method (p. [1002](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec50_html#page_1002)) to acquire the lock. Once a Lock has been obtained by one thread, the Lock will not allow another thread to obtain it until the first thread releases it (by calling the Lock’s unlock method; p. [1002](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec50_html#page_1002)).

• If several threads are trying to call method lock on the same Lock object at the same time, only one thread can obtain the lock—the others are placed in the waiting state. When a thread callsunlock, the object’s lock is released and a waiting thread attempting to lock the object proceeds.

• Class ReentrantLock (p. [1002](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec50_html#page_1002)) is a basic implementation of the Lock interface.

• The ReentrantLock constructor takes a boolean that specifies whether the lock has a fairness policy (p. [1002](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec50_html#page_1002)). If true, the ReentrantLock’s fairness policy is “the longest-waiting thread will acquire the lock when it’s available”—this prevents indefinite postponement. If the argument is set to false, there’s no guarantee as to which waiting thread will acquire the lock when it’s available.

• If a thread that owns a Lock determines that it cannot continue with its task until some condition is satisfied, the thread can wait on a condition object (p. [1003](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec51_html#page_1003)). Using Lock objects allows you to explicitly declare the condition objects on which a thread may need to wait.

• Condition (p. [1003](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec51_html#page_1003)) objects are associated with a specific Lock and are created by calling Lockmethod newCondition, which returns a Condition object. To wait on a Condition, the thread can call the Condition’s await method. This immediately releases the associated Lock and places the thread in the waiting state for that Condition. Other threads can then try to obtain the Lock.

• When a runnable thread completes a task and determines that a waiting thread can now continue, the runnable thread can call Condition method signal to allow a thread in that Condition’swaiting state to return to the runnable state. At this point, the thread that transitioned from thewaiting state to the runnable state can attempt to reacquire the Lock.

• If multiple threads are in a Condition’s waiting state when signal is called, the default implementation of Condition signals the longest-waiting thread to transition to the runnablestate.

• If a thread calls Condition method signalAll, then all the threads waiting for that condition transition to the runnable state and become eligible to reacquire the Lock.

• When a thread is finished with a shared object, it must call method unlock to release the Lock.

• Locks allow you to interrupt waiting threads or to specify a timeout for waiting to acquire a lock—not possible with synchronized. Also, a Lock object is not constrained to be acquired and released in the same block of code, which is the case with the synchronized keyword.

• Condition objects allow you to specify multiple conditions on which threads may wait. Thus, it’s possible to indicate to waiting threads that a specific condition object is now true by calling thatCondition object’s signal or signallAll methods (p. [1003](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec51_html#page_1003)). With synchronized, there’s no way to explicitly state the condition on which threads are waiting.

#### Section 23.11 Multithreading with GUI: SwingWorker

• The event dispatch thread (p. [1011](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec11_html#page_1011)) handles interactions with the application’s GUI components. All tasks that interact with the GUI are placed in an event queue and executed sequentially by this thread.

• Swing GUI components are not thread safe. Thread safety is achieved by ensuring that Swing components are accessed from only the event dispatch thread.

• Performing a lengthy computation in response to a user interface interaction ties up the event dispatch thread, preventing it from attending to other tasks and causing the GUI components to become unresponsive. Long-running computations should be handled in separate threads.

• You can extend generic class SwingWorker (package javax.swing; p. [1011](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec11_html#page_1011)), which implementsRunnable, to perform a task in a worker thread then update Swing components from the event dispatch thread based on the task’s results. You override its doInBackground and done methods. Method doInBackground performs the computation and returns the result. Method done displays the results in the GUI.

• Class SwingWorker’s first type parameter indicates the type returned by the doInBackgroundmethod; the second indicates the type that’s passed between the publish and process methods to handle intermediate results.

• Method doInBackground is called from a worker thread. After doInBackground returns, methoddone is called from the event dispatch thread to display the results.

• An ExecutionException is thrown if an exception occurs during the computation.

• SwingWorker method publish repeatedly sends intermediate results to method process, which displays the results in a GUI component. Method setProgress updates the progress property.

• Method process executes in the event dispatch thread and receives data from method publish. The passing of values between publish in the worker thread and process in the event dispatch thread is asynchronous; process is not necessarily invoked for every call to publish.

• PropertyChangeListener (p. [1024](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec59_html#page_1024)) is an interface from package java.beans that defines a single method, propertyChange. Every time method setProgress is invoked, aPropertyChangeEvent is generated to indicate that the progress property has changed.

#### Section 23.12 Timing sort and parallelSort with the Java SE 8 Date/Time API

• Class Instant’s static method now gets the current time.

• To determine the difference between two Instants, use class Duration’s static methodbetween, which returns a Duration object containing the time difference.

• Duration method toMillis returns the Duration as a long value milliseconds.

• NumberFormat static method getPercentInstance returns a NumberFormat that’s used to format a number as a percentage.

• NumberFormat method format returns a String representation of its argument in the specified numeric format.

• Arrays static method parallelSetAll fills an array with values produced by a generator function that receives an int and returns a value of type int, long or double. Depending on which overload of method parallelSetAll is used the generator function is an object of a class that implements IntToDoubleFunction (for double arrays), IntUnaryOperator (for int arrays),IntToLongFunction (for long arrays) or IntFunction (for arrays of any non-primitive type).

• Arrays static method parallelPrefix applies a BinaryOperator to the current and previous array elements and stores the result in the current element.

#### Section 23.13 Java SE 8: Sequential vs. Parallel Streams

• Streams are easy to parallelize, enabling programs to benefit from enhanced performance on multi-core systems.

• To obtain a parallel stream, simply invoke method parallel on an existing stream.

#### Section 23.14 (Advanced) Interfaces Callable and Future

• The Callable (p. [1030](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec67_html#page_1030)) interface (of package java.util.concurrent) declares a single method named call that allows a task to return a value.

• ExecutorService method submit (p. [1030](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec67_html#page_1030)) executes a Callable passed in as its argument. Method submit returns an object of type Future (of package java.util.concurrent) that represents the future result of the executing Callable.

• Interface Future (p. [1030](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec67_html#page_1030)) declares method get to return the result of the Callable. The interface also provides methods that enable you to cancel a Callable’s execution, determine whether the Callable was cancelled and determine whether the Callable completed its task.

• Java SE 8 introduces a new CompletableFuture class (package java.util.concurrent; p.[1030](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec67_html#page_1030)), which implements the Future interface and enables you to asynchronously executeRunnables that perform tasks or Suppliers that return values.

• Interface Supplier (p. [1030](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec67_html#page_1030)), like interface Callable, is a functional interface with a single method (in this case, get) that receives no arguments and returns a result.

• CompletableFuture static method supplyAsync (p. [1033](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec68_html#page_1033)) asynchronously executes a Suppliertask that returns a value.

• CompletableFuture static method runAsync (p. [1034](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec72_html#page_1034)) asynchronously executes a Runnabletask that does not return a result.

• CompletableFuture method get is a blocking method—it causes the calling thread to wait until the asynchronous task completes and returns its results.

#### Section 23.15 (Advanced) Fork/Join Framework

• Java’s concurrency APIs include the fork/join framework, which helps programmers parallelize algorithms. The fork/join framework particularly well suited to divide-and-conquer-style algorithms, like the merge sort.

### Self-Review Exercises

[**23.1**](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec19_html#ch23ans1) Fill in the blanks in each of the following statements:

a) A thread enters the terminated state when \_\_\_\_\_\_\_\_\_.

b) To pause for a designated number of milliseconds and resume execution, a thread should call method \_\_\_\_\_\_\_\_\_ of class \_\_\_\_\_\_\_\_\_.

c) A runnable thread can enter the \_\_\_\_\_\_\_\_\_ state for a specified interval of time.

d) At the operating-system level, the runnable state actually encompasses two separate states, \_\_\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_\_.

e) Runnables are executed using a class that implements the \_\_\_\_\_\_\_\_\_ interface.

f) ExecutorService method \_\_\_\_\_\_\_\_\_ ends each thread in an ExecutorService as soon as it finishes executing its current Runnable, if any.

g) In a(n) \_\_\_\_\_\_\_\_\_ relationship, the \_\_\_\_\_\_\_\_\_ generates data and stores it in a shared object, and the \_\_\_\_\_\_\_\_\_ reads data from the shared object.

h) Keyword \_\_\_\_\_\_\_\_\_ indicates that only one thread at a time should execute on an object.

[**23.2**](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec19_html#ch23ans2) **(Advanced Optional Sections)** Fill in the blanks in each of the following statements:

a) Method \_\_\_\_\_\_\_\_\_ of class Condition moves a single thread in an object’s waiting state to therunnable state.

b) Method \_\_\_\_\_\_\_\_\_ of class Condition moves every thread in an object’s waiting state to therunnable state.

c) A thread can call method \_\_\_\_\_\_\_\_\_ on a Condition object to release the associated Lock and place that thread in the \_\_\_\_\_\_\_\_\_ state.

d) Class \_\_\_\_\_\_\_\_\_\_ implements the BlockingQueue interface using an array.

e) Class Instant’s static method \_\_\_\_\_\_\_\_\_ gets the current time.

f) Duration method \_\_\_\_\_\_\_\_\_ returns the Duration as a long value milliseconds.

g) NumberFormat static method \_\_\_\_\_\_\_\_ returns a NumberFormat that’s used to format a number as a percentage.

h) NumberFormat method \_\_\_\_\_\_\_\_\_ returns a String representation of its argument in the specified numeric format.

i) Arrays static method \_\_\_\_\_\_\_\_\_ fills an array with values produced by a generator function.

j) Arrays static method \_\_\_\_\_\_\_\_\_ applies a BinaryOperator to the current and previous array elements and stores the result in the current element.

k) To obtain a parallel stream, simply invoke method \_\_\_\_\_\_\_\_\_ on an existing stream.

l) Among its many features a CompletableFuture enables you to asynchronously execute \_\_\_\_\_\_\_\_\_ that perform tasks or \_\_\_\_\_\_\_\_\_ that return values.

[**23.3**](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec19_html#ch23ans3) State whether each of the following is true or false. If false, explain why.

a) A thread is not runnable if it has terminated.

b) Some operating systems use timeslicing with threads. Therefore, they can enable threads to preempt threads of the same priority.

c) When the thread’s quantum expires, the thread returns to the running state as the operating system assigns it to a processor.

d) On a single-processor system without timeslicing, each thread in a set of equal-priority threads (with no other threads present) runs to completion before other threads of equal priority get a chance to execute.

[**23.4**](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec19_html#ch23ans4) **(Advanced Optional Sections)** State whether each of the following is true or false. If false, explain why.

a) To determine the difference between two Instants, use class Duration’s static methoddifference, which returns a Duration object containing the time difference.

b) Streams are easy to parallelize, enabling programs to benefit from enhanced performance on multi-core systems.

c) Interface Supplier, like interface Callable, is a functional interface with a single method that receives no arguments and returns a result.

d) CompletableFuture static method runAsync asynchronously executes a Supplier task that returns a value.

e) CompletableFuture static method supplyAsync asynchronously executes a Runnable task that does not return a result.

### Answers to Self-Review Exercises

[**23.1**](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec18_html#ch23que1)

a) its run method ends.

b) sleep, Thread.

c) timed waiting.

d) ready, running.

e) Executor.

f) shutdown.

g) producer/consumer, producer, consumer.

h) synchronized.

[**23.2**](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec18_html#ch23que2)

a) signal.

b) signalAll.

c) await, waiting.

d) ArrayBlockingQueue.

e) now.

f) toMillis.

g) getPercentInstance.

h) format.

i) parallelSetAll.

j) parallelPrefix.

k) parallel.

l) Runnables, Suppliers.

[**23.3**](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec18_html#ch23que3)

a) True.

b) False. Timeslicing allows a thread to execute until its timeslice (or quantum) expires. Then other threads of equal priority can execute.

c) False. When a thread’s quantum expires, the thread returns to the ready state and the operating system assigns to the processor another thread.

d) True.

[**23.4**](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec18_html#ch23que4)

a) False. The Duration method for calculating the difference between two Instants is namedbetween.

b) True.

c) True.

d) False. The method that asynchronously executes a Supplier is supplyAsync.

e) False. The method that asynchronously executes a Runnable is runAsync.

### Exercises

**23.5****(True or False)** State whether each of the following is true or false. If false, explain why.

a) Method sleep does not consume processor time while a thread sleeps.

b) Swing components are thread safe.

c) **(Advanced)** Declaring a method synchronized guarantees that deadlock cannot occur.

d) **(Advanced)** Once a ReentrantLock has been obtained by a thread, the ReentrantLock object will not allow another thread to obtain the lock until the first thread releases it.

**23.6 (Multithreading Terms)** Define each of the following terms.

a) thread

b) multithreading

c) runnable state

d) timed waiting state

e) preemptive scheduling

f) Runnable interface

g) producer/consumer relationship

h) quantum

**23.7 (Advanced: Multithreading Terms)** Discuss each of the following terms in the context of Java’s threading mechanisms:

a) synchronized

b) wait

c) notify

d) notifyAll

e) Lock

f) Condition

**23.8 (Blocked State)** List the reasons for entering the blocked state. For each of these, describe how the program will normally leave the blocked state and enter the runnable state.

**23.9 (Deadlock and Indefinite Postponement)** Two problems that can occur in systems that allow threads to wait are deadlock, in which one or more threads will wait forever for an event that cannot occur, and indefinite postponement, in which one or more threads will be delayed for some unpredictably long time. Give an example of how each of these problems can occur in multithreaded Java programs.

**23.10 (Bouncing Ball)** Write a program that bounces a blue ball inside a JPanel. The ball should begin moving with a mousePressed event. When the ball hits the edge of the JPanel, it should bounce off the edge and continue in the opposite direction. The ball should be updated using a Runnable.

**23.11 (Bouncing Balls)** Modify the program in [Exercise 23.10](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec20_html#ch23que10) to add a new ball each time the user clicks the mouse. Provide for a minimum of 20 balls. Randomly choose the color for each new ball.

**23.12 (Bouncing Balls with Shadows)** Modify the program in [Exercise 23.11](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec20_html#ch23que11) to add shadows. As a ball moves, draw a solid black oval at the bottom of the JPanel. You may consider adding a 3-D effect by increasing or decreasing the size of each ball when it hits the edge of the JPanel.

**23.13 (Advanced: Circular Buffer with** ***Lock*s and** ***Condition*s)** Reimplement the example in[Section 23.8](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec8_html#ch23lev1sec8) using the Lock and Condition concepts presented in [Section 23.9](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec9_html#ch23lev1sec9).

**23.14 (Bounded Buffer: A Real-World Example)** Describe how a highway off-ramp onto a local road is a good example of a producer/consumer relationship with a bounded buffer. In particular, discuss how the designers might choose the size of the off-ramp.

#### Parallel Streams

For [Exercises 23.15](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec89_html#ch23que15)–[23.17](http://proquest.safaribooksonline.com/9780133813036/ch23lev2sec89_html#ch23que17), you may need to create larger data sets to see a significant performance difference.

**23.15****(Summarizing the Words in a File)** Reimplement [Fig. 17.17](http://proquest.safaribooksonline.com/9780133813036/ch17lev1sec7_html#ch17fig17) using parallel streams. Use the Date/Time API timing techniques you learned in [Section 23.12](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec12_html#ch23lev1sec12) to compare the time required for the sequential and parallel versions of the program.

**23.16 (Summarizing the Characters in a File)** Reimplement [Exercise 17.9](http://proquest.safaribooksonline.com/9780133813036/ch17lev1sec16_html#ch17que9) using parallel streams. Use the Date/Time API timing techniques you learned in [Section 23.12](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec12_html#ch23lev1sec12) to compare the time required for the sequential and parallel versions of the program.

**23.17 (Summarizing the File Types in a Directory)** Reimplement [Exercise 17.10](http://proquest.safaribooksonline.com/9780133813036/ch17lev1sec16_html#ch17que10) using parallel streams. Use the Date/Time API timing techniques you learned in [Section 23.12](http://proquest.safaribooksonline.com/9780133813036/ch23lev1sec12_html#ch23lev1sec12) to compare the time required for the sequential and parallel versions of the program.