Concurrency: Part II 23

Chapter Topics

- Understanding the inheritance hierarchy of the main interfaces of the Executor Framework: `Executor`, `ExecutorService`, and `Scheduled-ExecutorService` interfaces
- Using time units for time durations defined by the `TimeUnit` enum type
- Using a thread-local pseudorandom number generator defined by the `ThreadLocalRandom` class
- Creating executor services and scheduled executor services using methods of the `Executors` utility class
- Understanding the lifecycle of an executor service: `RUNNING`, `SHUTDOWN`, and `TERMINATION` states
- Controlling shutdown and termination of an executor service
- Defining tasks that do not return a result and those that do using the `Runnable` and the `Callable<V>` functional interfaces, respectively
- Submitting and invoking tasks using an executor service
- Handling task results using the `Future<V>` interface
- Scheduling tasks with a scheduled executor service
- Understanding task cancellation in an executor service
- Understanding the basics of solving divide-and-conquer problems using the Fork/Join Framework
- Understanding the implications of immutability
- Using the keyword `volatile` for visibility of shared fields to avoid memory contention errors
- Using atomic variables for mutually exclusive operations on shared single fields
- Using programmatic locking provided by reentrant locks and reentrant read-write locks to implement mutually exclusive operations on shared resources
- Using special-purpose synchronizers, such as the cyclic barrier and the count-down latch, to control thread execution behavior
- Using synchronized collections and maps for mutually exclusive operations on collections
- Using concurrent collections, concurrent maps, blocking queues, and copy-on-write collections for mutually exclusive operations on collections and maps

Java SE 17 Developer Exam Objectives

[8.1] Create worker threads using Runnable and Callable, manage the thread lifecycle, including automations provided by different

§23.2, p. 1423

Executor services and concurrent API ❍ *Creating worker threads with the* `Callable` *functional interface, and managing concurrency using executor services of the Concurrency API are covered in this chapter.* ❍ *For the* `Runnable` *functional interface and the thread lifecycle, see* §22.3, p. 1371, *and* §22.4, p. 1380.	
[8.2] Develop thread-safe code, using different locking mechanisms and concurrent API	§23.4, p. 1451 *to* §23.7, p. 1482
[8.3] Process Java collections concurrently including the use of parallel streams ❍ *Processing collections concurrently is covered in this chapter.* ❍ *For parallel streams, see* §16.9, p. 1009.	§23.7, p. 1482
Java SE 11 Developer Exam Objectives	
[8.1] Create worker threads using Runnable and Callable, and manage concurrency using an ExecutorService and java.util.concurrent API ❍ *Creating worker threads with the* `Callable` *functional interface, and managing concurrency using executor services of the Concurrency API are covered in this chapter.* ❍ *For the* `Runnable` *functional interface, see* §22.3, p. 1371.	§23.2, p. 1423
[8.2] Develop thread-safe code, using different locking mechanisms and java.util.concurrent API	§23.4, p. 1451 *to* §23.7, p. 1482

Low-level support for creating, starting, and coordinating threads in multithreaded applications is discussed in Chapter 22, p. 1365. This low-level approach to enforce correct execution order for concurrent threads inherently increases code complexity, potentially impacts performance, and is not always scalable. This chapter deals with high-level support for multithreaded programming that alleviates many of these problems in designing multithreaded applications. This support is primarily provided by the `java.util.concurrent` package and its subpackages: The Executor Framework, the

Fork/Join Framework, locking mechanisms, lock-free algorithms, and concurrent collections.

23.1 Utility Classes `TimeUnit` and `ThreadLocalRandom`

This section can be skipped on the first reading, and can be read if and when the need arises while working through this chapter.

Before diving into high-level concurrency support provided by the `java.util.concurrent` package, we begin by providing an overview of two utility classes whose methods are used in many of the examples presented in this chapter.

The `TimeUnit` Enum Type

The enum type `java.util.concurrent.TimeUnit` represents *time units* at different levels of granularity, from nanoseconds to days, as shown in **Table 23.1**.

A `TimeUnit` can be used to indicate the time unit in which a numerical value should be interpreted as a time duration. Many time-based methods interpret their timing parameter according to the time unit specified by this enum type. In the code below, the numerical value 2 will be interpreted by the `awaitTermination()` method as two seconds.

Click here to view code image

```
boolean allTerminated = executor.awaitTermination(2, TimeUnit.SECONDS);
```

Note that a `TimeUnit` enum constant does not maintain any time information, like date or time of day.

Table 23.1 *Constants Defined by the* `java.util.concurrent.TimeUnit` *Enum Type*

Constants defined by java.util.concurrent.TimeUnit enum type	Description of the time unit
`NANOSECONDS`	Representing one-thousandth of a microsecond
`MICROSECONDS`	Representing one-thousandth of a millisecond
`MILLISECONDS`	Representing one-thousandth of a second

Constants defined by java.util.concurrent.TimeUnit enum type	Description of the time unit
`MINUTES`	Representing 60 seconds
`SECONDS`	Representing 1 second
`HOURS`	Representing 60 minutes
`DAYS`	Representing 24 hours

The enum type `TimeUnit` also provides convenience methods that call the methods `Thread.sleep()`, `Thread.join()`, and `Object.wait()`, allowing the timeout for these methods to be specified in different time units. For example, the following method calls are equivalent:

Click here to view code image

```
TimeUnit.SECONDS.sleep(1);              // 1 second.
TimeUnit.MILLISECONDS.sleep(1000);      // 1000 milliseconds.
Thread.sleep(1000);                     // 1000 milliseconds.
```

Following convenience methods are defined in the `TimeUnit` enum type:

Click here to view code image

```
void sleep(long timeout)                    throws InterruptedException
void timedJoin(Thread thread, long timeout) throws InterruptedException
void timedWait(Object obj, long timeout)    throws InterruptedException
```

Call the `Thread.sleep(long millis)` method, the `Thread.join(long millis)` method, and the `Object.wait(long millis)` method, respectively, where the specified `timeout` is converted to milliseconds using this time unit.

The `ThreadLocalRandom` Class

Some examples in this chapter make use of a *pseudorandom generator*. The class `java.util.concurrent.ThreadLocalRandom` provides a pseudorandom generator that is confined to the current thread—that is, it is local to the thread. The `ThreadLocalRandom` class extends the `java.util.Random` class, and provides methods that return uniformly distributed numerical values. The `ThreadLocalRandom` class is particularly recommended

for concurrent tasks, instead of using a shared object of the `Random` class, as it entails less overhead and resource contention.

The code below illustrates use of the class in a task that simulates a dice. The `current()` static method of the class returns the `ThreadLocalRandom` object associated with the current thread.

Click here to view code image

```
int diceValue = ThreadLocalRandom.current().nextInt(1, 7); // [1, 6]
```

In addition, the `ThreadLocalRandom` class provides methods that return numerical streams whose elements are uniformly distributed pseudorandom numbers. The code below creates an `int` stream of 100 randomly generated `int` values which are in the interval `[1, 6]`, and counts the number of times the dice value `6` occurs in the stream.

Click here to view code image

```
long count = ThreadLocalRandom.current()
                              .ints(100, 1, 7).filter(i -> i == 6).count();
```

Following selected methods are defined in the `ThreadLocalRandom` enum type:

Click here to view code image

```
static ThreadLocalRandom current()
```

Returns the `ThreadLocalRandom` object associated with the current thread.

Click here to view code image

```
int nextInt(int bound)
int nextInt(int origin, int bound)
```

The two methods return a pseudorandom, uniformly distributed `int` value between `[0, bound)` and `[origin, bound)`, respectively, where `[n, m)` defines a half-open interval, where the end value is excluded.

The class `ThreadLocalRandom` also defines the analogous methods `nextLong()`, `nextFloat()`, and `nextDouble()` for the primitive types `long`, `float`, and `double`, respectively.

Click here to view code image

```
IntStream ints()
IntStream ints(long streamSize)
IntStream ints(int randomNumberOrigin, int randomNumberBound)
IntStream ints(long streamSize, int randomNumberOrigin,
               int randomNumberBound)
```

Each method returns a stream of pseudorandom `int` values.

The first method returns an unlimited number of pseudorandom `int` values in the stream, which can be any legal `int` value.

The second method returns `streamSize` pseudorandom `int` values in the stream, which can be any legal `int` value.

The third method returns unlimited pseudorandom `int` values in the stream, which are in the half-open interval `[randomNumberOrigin, randomNumberBound)`.

The fourth method returns `streamSize` pseudorandom `int` values in the stream, which are in the half-open interval `[randomNumberOrigin, randomNumberBound)`.

The class `ThreadLocalRandom` also defines the analogous methods `longs()` and `doubles()` for the primitive types `long` and `double` that return `LongStream` and `DoubleStream`, respectively.

23.2 The Executor Framework

Executors provide a high-level approach to launching tasks and managing threads. Tasks are executed asynchronously by threads in a multithreaded environment, where a *task* defines a unit of work—that is, a task is a set of instructions executed by a thread.

Internally, an executor maintains a *thread pool* that utilizes a number of reusable threads that are assigned tasks from a task queue, according to the execution policy of the executor. The threads maintained by a thread pool are also known as *worker threads*. A worker thread remains alive after executing a task, and can be assigned another task that is waiting to be executed.

Executors allow the *submission of tasks* for execution to be decoupled from the actual *execution of tasks*. The application defines and submits the tasks to the executor which manages their execution by assigning them to threads. The executor provides the necessary support for managing its lifecycle that comprises creating the executor, submitting tasks, managing the outcome of task execution, and shutting down the executor.

Figure 23.1 shows the interfaces in the `java.util.concurrent` package that define the characteristics of executors from basic to more advanced executors. The `Executors` util-

ity class provides static methods that create various kinds of executors that not only implement the executor interfaces shown in **Figure 23.1**, but also implement various policies for task execution (**Table 23.2**, **p. 1426**).

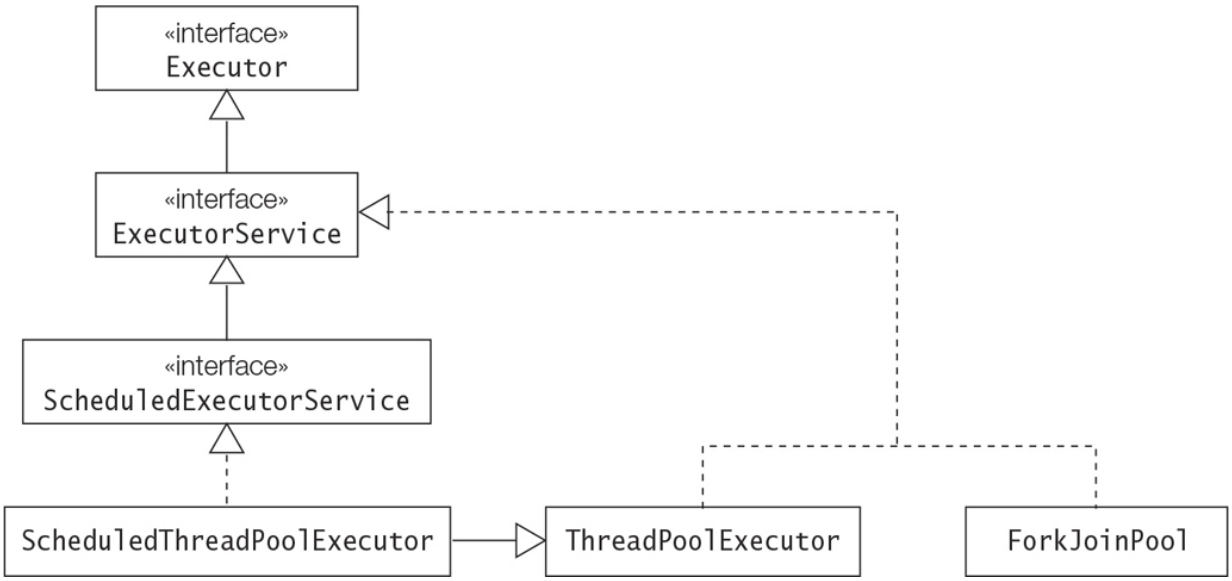

Figure 23.1 *Executor Interfaces in the* `java.util.concurrent` *Package*

The `Executor` Interface

The `java.util.concurrent.Executor` interface defines a basic executor. This executor executes a `Runnable` task that is passed to its `execute()` method. Technically, the `Executor` interface is a functional interface as it has a single abstract method. However, it is not annotated as such in the Java SE Platform API documentation, as it is recommended to use the more versatile executors that extend the `Executor` interface (**Figure 23.1**).

The `SimpleExecutor` class below implements the `Executor` interface. Its `execute()` method shows two implementations, (1a) and (1b), for executing a `Runnable` task. Typically, an executor delegates the task to a new thread for asynchronous execution as shown at (1a). It can also be executed synchronously by calling the `run()` method on the task as shown at (1b).

Click here to view code image

```
class SimpleExecutor implements Executor {
  @Override
  public void execute(Runnable task) {

    new Thread(task).start();         // (1a) Asynchronous call
//  task.run();                       // (1b) Synchronous call. Not recommended.
  }
}
// Client code
SimpleExecutor executor = new SimpleExecutor();
```

```
Runnable task = () -> System.out.println("Executing task ...");
executor.execute(task);
```

The essence of the basic executor is to replace the following code for running a task in a thread:

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

with the following code using an executor, where the actual execution is managed by the executor, abstracting away the management of the underlying thread:

```
executor.execute(task);
```

Typically, the tasks are executed by a *thread pool* maintained by the executors provided by the `Executors` utility class.

The following method is defined in the `Executor` interface:

```
void execute(Runnable task)
```

Executes the given `task` at some time in the future. The task may execute in a new thread, in a worker thread, or in the calling thread, at the discretion of the `Executor` implementation. Note that this method does not return a value and it does not specify any `throws` clause with checked exceptions.

The `Executors` Class

The `java.util.concurrent.Executors` class provides factory and utility methods for the interfaces defined in the `java.util.concurrent` package. In particular, it provides factory methods that return versatile implementations of the `ExecutorService` (**p. 1427**) and the `ScheduledExecutorService` (**p. 1440**) interfaces shown in **Figure 23.1**. The executor services created by the methods of the `Executors` class are summarized in **Table 23.2**, showing the respective static methods, the task execution policy of the executor services they create, and examples where they are used. For many applications, these executor services will be adequate, and examples in this chapter make heavy use of them.

In addition, the `java.util.concurrent.ThreadPoolExecutor` class, shown in **Figure 23.1**, provides constructors to create even more versatile executors with various properties such as minimum and maximum pool size, keep-alive time for idle threads, a blocking queue for managing tasks, and a handler for tasks that cannot be executed.

Table 23.2 *Selected Executor Services Provided by the* `Executors` *Utility Class*

The static method that creates an executor that implements the `ExecutorService` interface (p. 1427), and the executor's task execution policy	Description of the `ExecutorService` implementation
Single Thread Executor `newSingleThreadExecutor()` *Task execution policy*: tasks executed sequentially by a single thread. *(**Example 23.5**, **p. 1446**)*	Provides a single worker thread to which tasks can be assigned. If this single thread terminates due to a failure, a new thread will take its place if needed. Tasks in the queue are guaranteed to execute sequentially. Logically it is an equivalent of a fixed thread pool of size 1, but it cannot be reconfigured to use additional threads.
Fixed Thread Pool `newFixedThreadPool(int nThreads)` *Task execution policy*: tasks executed concurrently by a fixed number of threads. *(**Example 23.1**, **p. 1429**)* *(**Example 23.2**, **p. 1438**)*	Provides a thread pool of fixed size. The number of threads actively processing tasks is, at the most, equal to the pool size. The number of tasks submitted can be greater than the pool size, but some tasks may have to wait in the queue for threads to become available within the pool. If a thread terminates due to a failure, a new thread will take its place if needed. Pool threads are reused as required to execute the submitted tasks. They remain blocked when not executing a task.

Work Stealing Pool
Click here to view code image
`newWorkStealingPool()`
`newWorkStealingPool(int parallelism)`
	Provides a thread pool that tries to maintain its *target parallelism level*—that is, the maximum number of threads executing, or those available to

The static method that creates an executor that implements the `ExecutorService` interface (p. 1427), and the executor's task execution policy	Description of the `ExecutorService` implementation
Task execution policy: tasks executed concurrently while attempting to maintain its target parallelism level.	execute tasks. The actual number of threads can vary dynamically. It makes no guarantees about the order in which the tasks will be executed. The first method uses the number of available processors as its target parallelism level.
Cached Thread Pool `newCachedThreadPool()` *Task execution policy*: tasks executed concurrently by a thread pool whose size can vary dynamically with the number of tasks remaining to execute.	Provides a thread pool whose size may vary dynamically. Submitted tasks reuse available threads in the pool. Additional threads are created as needed when no threads are available. Unused threads eventually time out and are removed from the cache, so a pool that remains idle for long enough will not consume any resources.
Single Thread Scheduled Executor Click here to view code image `newSingleThreadScheduledExecutor()` *Task execution policy*: tasks executed sequentially by a single thread according to the schedule policy specified for each task. *(Example 23.5, p. 1446)*	Provides a single worker thread that works very much like a combination of a scheduled thread pool of size 1 and a single thread executor. It can execute submitted tasks sequentially, using a single thread, but with a specified delay; or periodically, based on a schedule. The schedule is specified individually for each task when it is submitted to the executor.
Scheduled Thread Pool	Provides a thread pool of a fixed size, that can execute

The static method that creates an executor that implements the `ExecutorService` interface (p. 1427), and the executor's task execution policy	Description of the `ExecutorService` implementation

`newScheduledThreadPool(`
`  int corePoolSize)`
 	
Task execution policy: tasks executed concurrently by a fixed number of threads, according to the schedule policy specified for each task.
(Example 23.3, p. 1441)
(Example 23.5, p. 1446)
	submitted tasks with a specified delay or periodically, based on a schedule. The schedule is specified individually for each task when it is submitted to the executor.

The `ExecutorService` Interface

Table 23.2 lists the executor services provided by the factory methods of the `Executors` class. An *executor service* implements the `ExecutorService` interface that extends the `Executor` interface with methods that provide the following functionality:

- Flexibly submitting tasks to the executor service and handling the results returned (p. 1436)
  In addition to the `execute()` method, an executor service provides the overloaded methods `submit()`, `invokeAll()`, and `invokeAny()` for submitting tasks. The `Future<V>` interface provides methods to handle the execution status and the results returned by tasks (p. 1435). The `Callable<V>` functional interface can be used to implement tasks that return a value (p. 1434).
- Managing the *executor lifecycle*
  The `ExecutorService` interface provides methods to manage the lifecycle of an executor (Figure 23.2). Once an executor service is created, for example, by static methods of the `Executors` class, it is in the `RUNNING` state, ready to accept tasks for execution. As an executor service is a precious resource, it must be shut down by calling its `shutdown()` or `shutdownNow()` method. An executor service in the `SHUTDOWN` state does not accept new tasks, which are discarded. However, although the executor service might be shut down, some of its threads might still be executing tasks and other tasks might be waiting to be executed. Only when all tasks have completed does the executor service transition to the `TERMINATED` state. The methods `isShutdown()` and `isTerminated()` can be used to check if the executor service is in the `SHUTDOWN` or `TERMINATED` state, respectively. The method `awaitTermination()`, called on an executor service after it is shut down, can be used to wait for threads to complete their tasks. See Example 23.1, p. 1429.

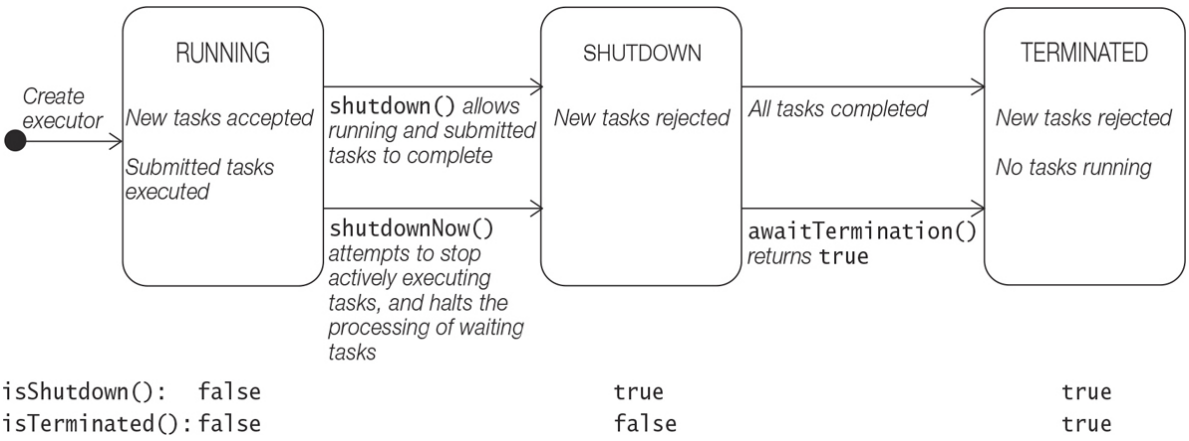

Figure 23.2 *Executor Service Lifecycle*

Following are selected methods of the `ExecutorService` interface for shutting down and terminating an executor service:

```
void shutdown()
```

Starts an orderly shutdown in which previously submitted tasks are executed, but new tasks are rejected. This method does *not* wait for already submitted tasks to complete execution. A shutdown request on an executor that is already shut down has no additional effect.

```
List<Runnable> shutdownNow()
```

Attempts to stop all actively executing tasks, and halts the processing of waiting tasks. It returns a list of the tasks that were awaiting execution.

This method does *not* wait for actively executing tasks to terminate.

This method at best attempts to stop processing of actively executing tasks— however, there are no guarantees. Typically, task cancellation will be triggered via `Thread.interrupt()`, so any task failing to respond to interrupts may never terminate (§22.4, p. 1393).

```
boolean isShutdown()
```

Returns `true` if this executor has been shut down—that is, a shutdown request has already been invoked on the executor.

Click here to view code image

```
boolean awaitTermination(long timeout, TimeUnit unit)
```

```
                                throws InterruptedException
```

Blocks until one of these events occurs first: All tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted. This method should be called after calling the `shutdown()` method.

```
boolean isTerminated()
```

Returns `true` if *all* tasks have completed following shutdown. It never returns `true` unless a shutdown request has already been invoked on the executor.

Using an Executor Service

The idiom for using an executor service is embodied in the following steps that are illustrated in **Example 23.1**:

• *Create the executor service.*

• *Submit tasks to the executor service.*

• *Shut down and terminate the executor service.*

Example 23.1 *Executor Lifecycle*

Click here to view code image

```
package executors;
import java.util.concurrent.*;

public class ExecutorLifecycle {

  // Task: dice roll.
  private static final Runnable diceRoll = () -> {                          // (1)
    int diceValue = ThreadLocalRandom.current().nextInt(1, 7);          // [1, 6]

    String threadName = Thread.currentThread().getName();
    System.out.println(threadName + " => dice value: " + diceValue);
    try {
      TimeUnit.MILLISECONDS.sleep(100);
    } catch (InterruptedException e) {
      System.out.println(threadName + ": " + e);
    }
  };

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

```
    System.out.printf("%50s   %s%n", "isShutdown()", "isTerminated()");
    // Create the executor service:
    ExecutorService es = Executors.newFixedThreadPool(3);                 // (2)
    try {                                                                 // (3)
      checkStates(es, "Before execute() at (4): ");

      // Submit tasks:
      es.execute(diceRoll);                                               // (4)
      es.execute(diceRoll);
      es.execute(diceRoll);
      checkStates(es, "After execute() at (4): ");
    } finally {                                                           // (5)
      // Shut down the executor service:
      checkStates(es, "Before shutdown() at (6a): ");
      es.shutdown();                                                      // (6a)
      checkStates(es, "After shutdown() at (6a): ");

//    checkStates(es, "Before shutdownNow() at (6b): ");
//    es.shutdownNow();                                                   // (6b)
//    checkStates(es, "After shutdownNow() at (6b): ");
    }

    // Second phase of shutdown:
//  awaitAndShutdownNow(es, 2, TimeUnit.SECONDS);                         // (7a)
//  awaitAndShutdownNow(es, 1, TimeUnit.MILLISECONDS);                    // (7b)
  }

  private static void checkStates(ExecutorService es, String msg) {      // (8)
    System.out.printf("%-40s %-5s           %-5s%n",
        msg, es.isShutdown(), es.isTerminated());
  }

  private static void awaitAndShutdownNow(                                // (9)
      ExecutorService es, int timeout, TimeUnit timeunit) {
    try {
      // Timed wait for tasks to complete execution:
      if (!es.awaitTermination(timeout, timeunit)) {                      // (10)
        // Attempt to cancel any uncompleted and waiting tasks:
        checkStates(es, "Before shutdownNow() at (11): " );
        es.shutdownNow();                                                 // (11)
        checkStates(es, "After shutdownNow() at (11): " );
        // Timed wait for tasks to be cancelled:
        while (!es.awaitTermination(timeout, timeunit)) {                 // (12)
          System.out.println("All tasks not yet completed at (12).");
        }
      }
      checkStates(es, "After awaitTermination() at (10): " );
    } catch (InterruptedException ie) {                                   // (13)
      // Attempt to cancel any uncompleted and waiting tasks:
      es.shutdownNow();                                                   // (14)
      // Reinstate the interrupt status.
```

```
            Thread.currentThread().interrupt();
            checkStates(es, "After interruption: ");
        }
      }
    }
```

Probable output from the program when calling the `shutdown()` method at (6a):

Click here to view code image

```
                                      isShutdown()   isTerminated()
Before execute() at (4):                 false           false
After execute() at (4):                  false           false
Before shutdown() at (6a):               false           false

pool-1-thread-3 => dice value: 6
pool-1-thread-1 => dice value: 2
pool-1-thread-2 => dice value: 1
After shutdown() at (6a):                true            false
```

Probable output from the program when calling the `shutdownNow()` method at (6b):

Click here to view code image

```
                                      isShutdown()   isTerminated()
Before execute() at (4):                 false           false
After execute() at (4):                  false           false
pool-1-thread-3 => dice value: 4
pool-1-thread-2 => dice value: 5
pool-1-thread-1 => dice value: 5
Before shutdownNow() at (6b):            false           false
After shutdownNow() at (6b):             true            false
pool-1-thread-3: java.lang.InterruptedException: sleep interrupted
pool-1-thread-1: java.lang.InterruptedException: sleep interrupted
pool-1-thread-2: java.lang.InterruptedException: sleep interrupted
```

Example 23.1 implements the necessary steps for using an executor service:

- *Create the executor service.*
  The utility class `Executors` provides methods for creating a wide range of executor services (**Table 23.2**). Appropriate executor service is created based on the desired task execution policy. In **Example 23.1**, a fixed thread pool with three worker threads is created at (2) to execute three tasks.
  Click here to view code image

```
ExecutorService es = Executors.newFixedThreadPool(3);
```

- *Submit tasks to the executor service.*
  The task submitted to the executor service is defined as `Runnable` at (1). It simulates a dice roll, and when executed, it prints the dice value. The executing thread then sleeps for 100 milliseconds to prolong the execution time for the task to illustrate executor shutdown and termination.
  The tasks are submitted to the executor service using the `execute()` method— other methods are explored later (**p. 1436**, **p. 1440**). Each call to the `execute()` method will result in a new task being submitted. The implementation of the `Runnable diceRoll` will be executed three times, each time as a separate task.
  Click here to view code image

```
es.execute(diceRoll);                                                    // (4)
es.execute(diceRoll);
es.execute(diceRoll);
```

- *Shut down and terminate the executor service.*
  When the executor service is no longer needed, it should be closed properly so that resources associated with the executor service can be reclaimed. As an executor service is not `AutoCloseable`, we cannot use the `try`-with-resources statement. The simplest shutdown procedure involves either calling the `shutdown()` or `shutdownNow()` method for this purpose. Calling either of these methods shuts down the executor service, after which any attempts to submit new tasks to the executor service will be rejected. However, the `shutdown()` method will allow actively executing tasks and any waiting tasks to complete, but the `shutdown-Now()` method will attempt to cancel actively executing tasks and halt the processing of any waiting tasks.
  The shutdown procedure is typically performed in a `finally` block associated with a `try` block containing the code that uses the executor service. This setup ensures that the executor service will always be shut down no matter how the tasks execute.
  The method `checkStates()` at (9) in **Example 23.1** is used to determine whether the executor service has been shut down and/or terminated at various stages in the execution, as can be seen in the program output.
  The code below from **Example 23.1** sketches the steps involved in using an executor:
  Click here to view code image

```
// Create the executor service:
ExecutorService es = Executors.newFixedThreadPool(3);
try {                                                                    // (3)
  // Submit tasks:
  es.execute(diceRoll);                                                  // (4)
  es.execute(diceRoll);
  es.execute(diceRoll);
} finally {                                                              // (5)
  // Shut down the executor service:
  es.shutdown();                                                         // (6a)
```

```
//es.shutdownNow();                                                    // (6b)
}
```

Two scenarios are shown in the output from the program in **Example 23.1**, corresponding to using the `shutdown()` and the `shutdownNow()` methods at (6a) and (6b), respectively. These methods do *not* wait for previously submitted tasks to complete execution before returning. Note that the `isTerminated()` method returns `false` after shutdown as one or more threads might not have completed execution—in this particular case, most probably waiting for sleep to time out. It is also important to note that the JVM will not terminate until all active threads in the thread pool have completed execution—analogous to normal threads in an application.

In the output, we see that the default name of a worker thread is composed of a thread pool number and a thread number—for example, `pool-1-thread-3`.

The output from the program when calling the `shutdown()` method at (6a) shows that the executor service did shut down, but the executor service did *not* terminate as some tasks continued execution—the method `isTerminated()` returned `false` when called after return from the `shutdown()` method. However, any running or submitted tasks were allowed to complete execution before the JVM terminated—in this particular case, threads were allowed to sleep until timed out.

In contrast, calling the `shutdownNow()` method cancels any executing and pending tasks. The `shutdownNow()` method does not wait for pending tasks to terminate.

From the output, we can see that calling the `isTerminated()` method on the executor returns `false`, as the executor has not terminated because all pending tasks have not been cancelled yet. The output from the program shows that, when cancelled, all three threads were interrupted while asleep. The `Interrupted-Exception` being caught and handled when the threads woke up and continued execution resulted in the tasks completing execution and the JVM being able to terminate.

Controlling Shutdown and Termination of an Executor Service

The `ExecutorService` interface provides the `awaitTermination()` method, which in conjunction with the shutdown methods, allows more refined control over pending tasks at shutdown.

Further control in shutting down and terminating an executor service is implemented by the `awaitAndShutdownNow()` method defined at (9) in **Example 23.1**. This method utilizes both the `awaitTermination()` method and the `shutdownNow()` method of the executor service. The executor service to shut down and the timeout to wait for pending tasks to complete are passed as parameters to the `awaitAndShutdownNow()` method.

The `awaitAndShutdownNow()` method first calls the `awaitTermination()` method at (10) that blocks until one of the following events occurs:

- The `awaitTermination()` method returns `true`, indicating that all tasks have completed, and thereby the executor service has been shut down and terminated.
- The `awaitTermination()` method returns `false`, indicating that there are still uncompleted tasks. The `shutdownNow()` method is called at (11) to cancel any uncompleted and waiting tasks. The `awaitTermination()` method is called again in a `while` loop at (12) to wait for any tasks still pending completion. This loop terminates when all tasks have completed; if the current thread is interrupted while awaiting task completion, the control goes to (13).
- The current thread is interrupted while waiting for the blocking call to the `awaitTermination()` method to return, in which case the `InterruptedException` thrown is caught by the `catch` block at (13). A further attempt is made in the `catch` block to cancel any pending tasks by calling the `shutdownNow()` method at (14).

The `awaitAndShutdownNow()` method at best makes an attempt at allowing pending tasks to complete after shutdown. It is called at (7a) and (7b) in **Example 23.1** to illustrate shutting down an executor service and controlling any of its pending tasks. (For this discussion, (6b) is commented out in **Example 23.1**.)

The duration of the wait for pending tasks in the call to the `awaitTermination()` method impacts the fate of the pending tasks. The following call:

Click here to view code image

```
awaitAndShutdownNow(es, 2, TimeUnit.SECONDS);                    // (7a)
```

can result in the following output from the program, showing that two seconds of waiting was sufficient for all tasks to complete execution at (10), causing the executor service to terminate:

Click here to view code image

```
                                        isShutdown()   isTerminated()
Before execute() at (4):                   false          false
After execute() at (4):                    false          false
Before shutdown() at (6a):                 false          false
pool-1-thread-2 => dice value: 4
pool-1-thread-3 => dice value: 1
pool-1-thread-1 => dice value: 2
After shutdown() at (6a):                  true           false
After awaitTermination() at (10):          true           true
```

However, the following call:

Click here to view code image

```
  awaitAndShutdownNow(es, 1, TimeUnit.MILLISECONDS);                    // (7b)
```

can result in the following output from the program, showing that a one-millisecond wait was insufficient for all tasks to complete execution at (10), resulting in the `shutdownNow()` method at (11) being called. Any pending tasks are thus interrupted and the `InterruptedException` is handled by each task. The `awaitTermination()` method returns `true` when called in the `while` loop at (12), indicating that the interrupted tasks completed their execution:

Click here to view code image

```
                                      isShutdown()   isTerminated()
Before execute() at (4):                 false           false
After execute() at (4):                  false           false
Before shutdown() at (6a):               false           false
pool-1-thread-2 => dice value: 3
pool-1-thread-1 => dice value: 1
pool-1-thread-3 => dice value: 1
After shutdown() at (6a):                true            false
Before shutdownNow() at (11):            true            false
After shutdownNow() at (11):             true            false
pool-1-thread-3: java.lang.InterruptedException: sleep interrupted
pool-1-thread-2: java.lang.InterruptedException: sleep interrupted
pool-1-thread-1: java.lang.InterruptedException: sleep interrupted
After awaitTermination() at (10):        true            true
```

Defining Tasks That Return a Result

Implementations of the `java.util.concurrent.Callable<V>` functional interface represent tasks that are designed to be executed by threads—analogous to implementations of the `java.lang.Runnable` functional interface. Whereas we can create and start a thread by passing a `Runnable` to a constructor of the `Thread` class, this is not possible with a `Callable<V>` implementation, as `Thread` constructors do not accept a `Callable<V>` object. In order to execute an implementation of the `Callable<V>` functional interface as a task in a thread, we can use an executor service—a topic which we explore thoroughly later (p. 1436).

In contrast to the `Runnable` interface, the `Callable<V>` interface is generic. As opposed to the void `run()` method of the `Runnable` interface, the `call()` method of the `Callable<V>` interface returns a result from the task it represents and can throw checked exceptions. As it is a functional interface, the implementation of its sole method `call()` can be provided by a lambda expression.

An executor service provides bulk execution of tasks grouped in a collection using its methods `invokeAny()` and `invokeAll()` (p. 1439). Such bulk tasks can only be imple-

mented as `Callable<V>` tasks, and not as `Runnable` tasks.

Typically, a `Runnable` is used for long-running tasks, such as a server, whereas a `Callable<V>` is a one-shot task that needs to compute and return a result.

The code at (1) implements a `Callable<Integer>` that simulates a dice using a `Random` object (§8.7, p. 482). The method `call()` is invoked directly on this implementation at (2) to roll the dice. Later we will see how we can obtain the value of a dice roll asynchronously in a thread using an executor service (p. 1438).

Click here to view code image

```
Random rng = new Random();
Callable<Integer> dice = () -> rng.nextInt(1,7);  // (1)
try {
  Integer diceValue = dice.call();     // (2)
  System.out.println(diceValue);       // Prints a value in the interval [1,6].
} catch (Exception ex) {
  ex.printStackTrace();
}
```

The following method is defined in the `Callable<V>` functional interface:

```
V call() throws Exception
```

Computes a result, or throws an exception if unable to do so.

Representing Task Results

A `Future<V>` object represents the result of a task that is asynchronously executed— that is, it represents the result of a task that is executed sometime in the future after its submission. For example, methods for submitting or scheduling tasks with an executor service return immediately. When a submitted task is executed sometime in the future, its result is returned in a `Future<V>` object that can be queried to obtain the result. A `Future<V>` object also provides methods to determine whether the task it represents completed normally or was cancelled (p. 1444).

The API of the `Future<V>` interface shown below provides references to examples that compute task results.

Click here to view code image

```
V get() throws InterruptedException, ExecutionException
V get(long timeout, TimeUnit unit)
      throws InterruptedException, ExecutionException, TimeoutException
```

The first method blocks if necessary for the computation to complete, and then retrieves the result contained in this `Future<V>` object. In other words, the method does not return unless the task completes execution.

The second method blocks if necessary at most for the duration of the specified `timeout` in order for the computation to complete, and then retrieves the result contained in this `Future<V>` object, if one is available.

Calling these methods on a computation that has been cancelled, throws a `java.util.concurrent.CancellationException`. See **Example 23.5**, **p. 1446**.

Click here to view code image

```
boolean cancel(boolean mayInterruptIfRunning)
```

Attempts to cancel the execution of the task this `Future<V>` object represents.

If this task is already completed or cancelled, or if could not be cancelled, the method has no effect. Otherwise, if this task has not started when the method is called, this task is never run.

If the task has already started, then the `mayInterruptIfRunning` parameter determines whether the thread executing this task is interrupted in an attempt to stop the task. If the `mayInterruptIfRunning` parameter is `true`, the thread executing the task is interrupted; otherwise, the task is allowed to complete.

This method returns `false` if the task could not be cancelled, typically because it has already completed; otherwise, it returns `true`. However, the return value from this method does not necessarily indicate whether the task is now cancelled. For that purpose, the `isCancelled()` method should be used. See **Example 23.5**, **p. 1446**.

```
boolean isCancelled()
```

Returns `true` if this task was *cancelled* before it completed normally. See **Example 23.5**, **p. 1446**.

```
boolean isDone()
```

Returns `true` if this task *completed*. Note that completion implies either normal termination, or an exception, or cancellation.

Submitting Tasks and Handling Task Results

Tasks executed by executor services are represented by implementations of either the `Runnable` or the `Callable<V>` functional interface. The following selected methods of the `ExecutorService` interface can be used for submitting tasks to an executor service:

Click here to view code image

```
Future<?> submit(Runnable task)
<V> Future<V> submit(Runnable task, V result)
<V> Future<V> submit(Callable<V> task)
```

All three methods submit a task for execution to the executor service. The first two methods submit a `Runnable` task, but the last one submits a value-returning `Callable<V>` task.

All three methods return a `Future<V>` object that represents the task. On *successful* execution of the task, the `isDone()` method of the `Future<V>` object will return `true`. The value returned in the `Future<V>` object can be obtained by calling the `get()` method on the `Future<V>` object. The `Future` object will contain the value `null`, or the specified `result` in the call, or the result of executing the value-returning `Callable<V>` task, respectively, for the three methods. See **Example 23.2**, **p. 1438**.

Click here to view code image

```
<V> List<Future<V>> invokeAll(Collection<? extends Callable<V>> tasks)
                              throws InterruptedException
<V> List<Future<V>> invokeAll(Collection<? extends Callable<V>> tasks,
                              long timeout, TimeUnit unit)
                              throws InterruptedException
```

Executes the given tasks and returns a list of `Future<V>` holding their status and results when *all* complete, or if the timeout expires first as in the second method. The `isDone()` method on each `Future<V>` object in the returned list of `Future<V>` instances will return `true`.

Upon return from the second method, any uncompleted tasks are cancelled through interruption. A *completed task* is one that has terminated either normally or by throwing an exception. The order of the `Future<V>` instances returned is the same as the

`Callable<V>` tasks in the collection passed as a parameter to the methods. See **Example 23.2, p. 1438**.

Click here to view code image

```
<V> V invokeAny(Collection<? extends Callable<V>> tasks)
                throws InterruptedException, ExecutionException
<V> V invokeAny(Collection<? extends Callable<V>> tasks,
                long timeout, TimeUnit unit)
                throws InterruptedException, ExecutionException,
                       TimeoutException
```

Executes the given tasks, returning the result of the first one that has *completed successfully without throwing an exception*, if any do succeed—or before the given timeout elapses in the case of the second method. Upon return, any uncompleted tasks are cancelled. See **Example 23.2, p. 1438**.

Submitting Tasks Individually

Example 23.2 illustrates submitting tasks *individually* and handling the task result. A `Runnable` task is defined at (1) that prints a dice value, and a `Callable<Integer>` task is defined at (2) that returns a dice value. We can of course call the `run()` and the `call()` methods to execute the `Runnable printDiceValue` and the `Callable<Integer> diceRoll`, respectively, but we would like to execute these tasks in a thread. For this purpose, a fixed thread pool is created at (3) in **Example 23.2**. Recall that this executor service reuses threads, and tasks may have to wait if all threads are occupied.

The overloaded `submit()` method of the `ExecutorService` interface can be used to submit tasks to the executor service. When a task submitted by these methods is executed asynchronously sometime in the future, the status or the result of executing the task is returned in a `Future<V>` object. The `get()` method of the `Future<V>` interface can be used to extract the value contained in a `Future<V>` object. However, note that the `get()` method will block the current thread until the task has completed. A timed version of the `get(timeout, timeunit)` method can also be used if there is a risk of waiting indefinitely.

The `Runnable printDiceValue` is submitted to the executor service at (4). The `get()` method, called at (5) to extract the value in the `Future<?>` object, blocks for the task to complete, if necessary. Since a `Runnable` does not return a value, the `get()` method returns the literal value `null` to indicate successful completion of the task. In this particular case, the type of the object returned is `Future<?>`, which is parameterized with the `?` wildcard to express that it contains an object of some unknown type.

The `Callable<Integer> diceValue` is submitted to the executor service at (6). The `Integer` result of executing this callable will be returned in a `Future<Integer>` object and can be extracted by calling the `get()` method on this object. The `get()` method blocks if the task has not completed execution.

Example 23.2 *Submitting Tasks and Handling Task Results*

Click here to view code image

```
package executors;
import java.util.List;
import java.util.concurrent.*;
import java.util.stream.Collectors;

public class TaskExecution {

  public static void main(String[] args) {
    // Runnable:
    Runnable printDiceValue = () ->                                    // (1)
      System.out.println("Execution of Runnable: "
                         + ThreadLocalRandom.current().nextInt(1,7));

    // Callable<V>:
    Callable<Integer> diceRoll =                                       // (2)
        () -> ThreadLocalRandom.current().nextInt(1,7);

    // Executor service:
    ExecutorService exs = Executors.newFixedThreadPool(3);             // (3)
    try {
      // Executing Runnable in a thread:
      Future<?> rfuture = exs.submit(printDiceValue);                  // (4)
      Object result = rfuture.get();                                   // (5)
      System.out.println("Result of Runnable: " + result);

      // Executing Callable<V> in a thread:
      Future<Integer> cfuture = exs.submit(diceRoll);                  // (6)
      Integer diceValue = cfuture.get();                               // (7)
      System.out.println("Result of Callable: " + diceValue);

      // Executing bulk tasks:
      List<Callable<Integer>> callables                                // (8)
          = List.of(diceRoll, diceRoll, diceRoll);

      List<Future<Integer>> allFutures = exs.invokeAll(callables);     // (9)
      List<Integer> resultList = allFutures.stream()                   // (10)
          .map(future -> {
            try {
              return future.get();
            } catch(InterruptedException | ExecutionException ie) {
```

```
                throw new IllegalStateException(ie);
              }
            })
            .toList();
        System.out.println("Result of invokeAll(): " + resultList);

        Integer anyResult = exs.invokeAny(callables);                    // (11)
        System.out.println("Result of invokeAny(): " + anyResult);
      } catch(InterruptedException | ExecutionException ie) {
        ie.printStackTrace();
      } finally {
        exs.shutdown();
      }
    }
  }
```

Probable output from the program:

Click here to view code image

```
Execution of Runnable: 4
Result of Runnable: null
Result of Callable: 2
Result of invokeAll(): [1, 3, 6]
Result of invokeAny(): 3
```

Invoking Tasks Collectively

Example 23.2 also illustrates invoking tasks *collectively* and handling the task results. An executor service defines the blocking methods `invokeAll()` and `invokeAny()` that allow a collection of `Callable<V>` tasks to be executed—thus implementing *bulk task execution*.

A list of `Callable<Integer>` tasks is defined at (8) in **Example 23.2**, where each task is represented by the `Callable<Integer> diceRoll` defined at (2). The list represents tasks for rolling three dice. This list is executed by calling the `invokeAll()` method of the executor service at (9). In contrast to the `submit()` method, the `invokeAll()` method *blocks* until *all* tasks have completed, and their results are returned in a list of `Future<Integer>` objects. The order of the `Future<Integer>` objects in the result list corresponds to the order of the tasks in the task list. The resulting list of `Future<Integer>` objects is processed in a stream at (10) to extract the result (i.e., an `Integer`) from each `Future<Integer>` object using the `get()` method, and the results are collected in a list of `Integer` that represents the results of throwing three dice.

On the other hand, the list of `Callable<Integer>` tasks is executed at (11) using the `invokeAny()` blocking method, which returns the result of the *first* task that completes successfully without throwing an exception. On return, any tasks not completed are can-

celled. Note that the `invokeAny()` method does *not* return the result in a `Future<Integer>` object.

Only `Callable<V>` tasks can be executed in bulk, as `Runnable` tasks cannot be passed as a parameter to the invoke methods.

The `ScheduledExecutorService` Interface

A `ScheduledExecutorService` is an `ExecutorService` that schedules tasks to be executed after a given delay or to be executed periodically. **Table 23.2** shows scheduled executor services provided by the factory methods of the `Executors` class.

The methods of the `ScheduledExecutorService` interface that schedule tasks return an object that implements the `ScheduledFuture<V>` interface. This interface extends the `Future<V>` interface, and thus provides methods to retrieve task results, to check execution status, or to cancel task execution.

Scheduling One-Shot Tasks

The overloaded `schedule()` method of the `ScheduledExecutorService` interface allows either a `Runnable` or a `Callable<V>` task to be scheduled for execution *only once after a specified delay*—that is, the `schedule()` methods execute *scheduled one-shot tasks*. The overloaded `schedule()` method of the `ScheduledExecutorService` interface is the analog of the overloaded `submit()` method of the `ExecutorService` interface for executing `Runnable` and `Callable<V>` tasks. These methods do not block—that is, they do not wait for the task to complete execution.

The following methods are defined in the `ScheduledExecutorService` interface that extends the `ExecutorService` interface:

Click here to view code image

```
ScheduledFuture<?> schedule(Runnable task, long delay, TimeUnit unit)
```

Schedules a one-shot task that becomes enabled after the given delay. The method returns a `ScheduledFuture<?>` object representing pending completion of the task and whose `get()` method will return `null` upon completion.

Click here to view code image

```
<V> ScheduledFuture<V> schedule(Callable<V> callable, long delay,
                                TimeUnit unit)
```

Schedules a value-returning one-shot task that becomes enabled after the given delay. The method returns a `ScheduledFuture<V>` object which can be used to extract the result computed by the scheduled task or to cancel the scheduled task.

Example 23.3 illustrates scheduling one-shot `Runnable` and `Callable<V>` tasks for execution using a scheduled executor service. The example uses a *scheduled* thread pool with two threads that is created at (1) by calling the `newScheduledThreadPool()` factory method of the `Executors` class.

The `Runnable` task defined at (2) is passed to the `schedule()` method of the scheduled executor service at (3) to be executed once after a delay of 500 milliseconds. The `get()` method invoked on the resulting `ScheduledFuture<?>` object returns the value `null`, as expected, after execution of the `Runnable` task to indicate successful completion.

The `Callable<String>` task defined at (4) is passed to the `schedule()` method of the scheduled executor service at (5) to be executed once after a delay of one second. In this case, the `get()` method invoked on the resulting `ScheduledFuture<String>` object returns a `String` value after successful completion of the task.

Example 23.3 *Executing Scheduled One-Shot Tasks*

Click here to view code image

```
package executors;
import java.util.concurrent.*;

public class ScheduledOneShotTasks {
  public static void main(String[] args) {
    ScheduledExecutorService ses = Executors.newScheduledThreadPool(2);  // (1)
    try {
      // Schedule a one-shot Runnable:
      Runnable runnableTask                                              // (2)
          = () -> System.out.println("I am a one-shot Runnable task.");
      ScheduledFuture<?> scheduledRunnableTaskFuture                     // (3)
          = ses.schedule(runnableTask, 500, TimeUnit.MILLISECONDS);
      System.out.println("Runnable result: " + scheduledRunnableTaskFuture.get());

      // Schedule a one-shot Callable<String>:
      Callable<String> callableTask
          = () -> "I am a one-shot Callable task.";                      // (4)
      ScheduledFuture<String> scheduledCallableTaskFuture
          = ses.schedule(callableTask, 1, TimeUnit.SECONDS);             // (5)
      System.out.println("Callable result: " + scheduledCallableTaskFuture.get());
    } catch (InterruptedException | ExecutionException exc) {
      exc.printStackTrace();
    } finally {
```

```
        ses.shutdown();
      }
    }
  }
```

Probable output from the program:

Click here to view code image

```
I am a one-shot Runnable task.
Runnable result: null
Callable result: I am a one-shot Callable task.
```

Scheduling Periodic Tasks

The `ScheduledExecutorService` interface provides two methods that can be used to schedule a `Runnable` task that should be executed periodically either after a fixed delay or at a fixed rate, with the first task execution commencing after an initial delay—in other words, scheduling *periodic tasks* for execution. No analogous methods are provided by the `ScheduledExecutorService` interface for `Callable<V>` tasks.

Click here to view code image

```
ScheduledFuture<?> scheduleWithFixedDelay(Runnable task, long initialDelay,
                                          long delay, TimeUnit unit)
ScheduledFuture<?> scheduleAtFixedRate(Runnable task, long initialDelay,
                                       long period, TimeUnit unit)
```

The `scheduleWithFixedDelay()` method schedules a periodic task that becomes enabled first after the given initial delay, and subsequently with the given `delay` *between the termination of one execution and the commencement of the next*. This means that the time interval between the *start* of two consecutive task executions is the sum of the execution time of the first task execution and the specified delay between them.

The `scheduleAtFixedRate()` method schedules a task that becomes enabled periodically for execution, with the first task execution commencing after the initial delay. This means that the time *between the start of two consecutive task executions is equal to the* `period`. Note that a new task execution commences after each period, regardless of whether or not the previous task execution has finished.

For both methods, task executions continue indefinitely, unless one of the following events occurs: The task is cancelled via the returned future, or the executor shuts down,

or an exception is thrown during task execution. The `isDone()` method on the returned future will then return `true`.

Example 23.4 illustrates scheduling `Runnable` tasks for repeated execution using a scheduled executor service. The `Runnable` task to be repeatedly executed is defined at (1). It prints information at the start and just before finishing its execution, which also includes the current time, by calling the auxiliary method `printTimestamp()` defined at (2). In between printing the information, the `Runnable` task sleeps for one second.

First scheduling a periodic task to be run with *a fixed delay* is illustrated in the `main()` method. An appropriate scheduled executor service is created at (3), namely, a scheduled thread pool of size 4. The method `scheduleWithFixedDelay()` is called at (4), passing the task, the initial delay of one second before commencement of the first task, and a fixed delay of three seconds between the end of one task execution and the start of the next one. The main thread sleeps for 15 seconds at (5) to allow the periodic task to be scheduled and executed. And finally the scheduled executor service is shut down at (6), which will allow the executing task to complete and the executor service to terminate. Without the shutdown, the periodic task would continue to be scheduled for execution. The output from the program shows that for the fixed delay, the delay between the finishing time of a task execution and the start time of the next one is approximately three seconds as it was specified—ignoring the nanoseconds in the current time.

Analogously, the scenario above is repeated for scheduling *a periodic task* to be executed at a fixed rate in the `main()` method. A new scheduled executor service is created at (7) for this purpose. This time the method `scheduleAtFixedRate()` is called at (8), passing the task, the initial delay of one second before commencement of the first task, and a fixed rate of three seconds between the *start* of each task execution.

The main thread sleeps for 10 seconds at (9) to allow the periodic task to be executed. The `shutdown()` method at (10) takes care of terminating the periodic task and the executor. The output from the program shows that for the fixed rate, the time between the start time of two consecutive task executions is approximately three seconds—again, ignoring the nanoseconds in the current time.

Example 23.4 *Executing Scheduled Periodic Tasks*

Click here to view code image

```
package executors;
import java.time.LocalTime;
import java.time.format.DateTimeFormatter;
import java.util.concurrent.*;
```

```
public class ScheduledPeriodicTasks {

  private static Runnable task = () -> {                                  // (1)
    printTimestamp(" Start:  I am on it!");
    try {
      TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }
    printTimestamp(" Finish: I am not on it!");
  };

  private static void printTimestamp(String msg) {                        // (2)
    String threadName = Thread.currentThread().getName();
    String ts = LocalTime.now().format(DateTimeFormatter.ofPattern("HH:mm:ss"));
    System.out.println(threadName + ": " + ts + msg);
  }

  public static void main(String[] args) {
    // Schedule a periodic task with fixed delay:
    ScheduledExecutorService ses1 = Executors.newScheduledThreadPool(4);  // (3)
    try {
      System.out.println("Fixed delay:");
      ses1.scheduleWithFixedDelay(task, 1, 3, TimeUnit.SECONDS);          // (4)
      TimeUnit.SECONDS.sleep(15);                                         // (5)
    } catch (InterruptedException e) {
      e.printStackTrace();
    } finally {
      ses1.shutdown();                                                    // (6)
    }

    // Schedule a periodic task at fixed rate:
    ScheduledExecutorService ses2 = Executors.newScheduledThreadPool(4);  // (7)
    try {
      System.out.println("\nFixed rate:");
      ses2.scheduleAtFixedRate(task, 1, 3, TimeUnit.SECONDS);             // (8)
      TimeUnit.SECONDS.sleep(10);                                         // (9)
    } catch (InterruptedException e) {
      e.printStackTrace();

    } finally {
      ses2.shutdown();                                                    // (10)
    }
  }
}
```

Probable output from the program:

Click here to view code image

```
Fixed delay:
pool-1-thread-1: 12:55:54 Start:  I am on it!
pool-1-thread-1: 12:55:55 Finish: I am not on it!
pool-1-thread-1: 12:55:58 Start:  I am on it!
pool-1-thread-1: 12:55:59 Finish: I am not on it!
pool-1-thread-2: 12:56:02 Start:  I am on it!
pool-1-thread-2: 12:56:03 Finish: I am not on it!
pool-1-thread-1: 12:56:06 Start:  I am on it!
pool-1-thread-1: 12:56:07 Finish: I am not on it!

Fixed rate:
pool-1-thread-2: 12:56:09 Start:  I am on it!
pool-1-thread-2: 12:56:10 Finish: I am not on it!
pool-1-thread-4: 12:56:12 Start:  I am on it!
pool-1-thread-4: 12:56:13 Finish: I am not on it!
pool-1-thread-4: 12:56:15 Start:  I am on it!
pool-1-thread-4: 12:56:16 Finish: I am not on it!
pool-1-thread-4: 12:56:18 Start:  I am on it!
pool-1-thread-4: 12:56:19 Finish: I am not on it!
```

Task Cancellation

Task cancellation is an important aspect of controlling task execution. Here we look at scenarios for cancelling tasks by invoking the `cancel()` method of the `Future<V>` object that represents the task after the task has been submitted to the executor. The effect of the `cancel()` method on the task depends on what stage the task is in when the `cancel()` method is called.

- *The task is in the queue, waiting to be assigned to a thread.*
  Calling the `cancel()` method on the `Future<V>` object that represents the task removes the task from the queue—thus cancelling the task.
- *The task is being executed by a thread.*
  Calling the `cancel()` method on the `Future<V>` object that represents the task sends an *interrupt signal* to the thread, the outcome of which depends on how the task handles the signal. In this section, we take a closer look at how cancellation can be organized programmatically when the thread that is executing a task is stopped by calling the `cancel()` method. See also the discussion on handling interrupt signals when a thread is interrupted by calling the `Thread.interrupt()` method (§22.4, p. 1393).
- *The task has completed execution.*
  Calling the `cancel()` method on the `Future<V>` object that represents the task has no effect on the task.
  Example 23.5 illustrates task cancellation by calling the `cancel()` method on the `Future<V>` object that represents the task.
  A computation-intensive task is defined by the `bigFactorial()` method at (1) to iteratively compute the factorial of a `BigInteger`. In each iteration of its `for(;;)` loop, the

method checks at (2) whether the thread executing the method has been interrupted. If that is the case, it throws an `InterruptedException`. This terminates the `for(;;)` loop. The exception is caught and handled by the `catch` block at (3), causing the method to return normally. If there is no interrupt signal, the `for(;;)` loop continues the factorial computation. It is important that the task defines an appropriate action when the thread is interrupted.

The `timedTaskCancellation()` method at (4) illustrates using the timed `get()` method at (7) to wait for the task submitted at (6) to either finish execution and print the result at (8), or time out and throw a `TimeoutException` if the task did not complete execution. The output from the program shows that a `TimeoutException` was thrown. This exception is caught by the `catch` block at (9). The task is cancelled in the `catch` block by calling the `cancel()` method on the `Future<BigInteger>` object at (10). Since the `cancel()` method at (10) is called with the value `true`, the thread executing the task represented by the future is interrupted in an attempt to stop the task. The `cancel()` method returns `true`, indicating that the thread was interrupted. Whether the task is now cancelled is determined by the calling the `isCancelled()` method of the `Future<BigInteger>` object at (11) that returns the value `true`, verifying the cancellation. As only a single task is executed in the scenario above, we have used a single thread executor that is created at (5). The output from this scenario depends on the execution time to compute the factorial for the specified number at (6) and the duration of the timeout in the `get()` method at (7).

The `scheduledTaskCancellation()` method at (13) illustrates how to cancel a submitted task by scheduling a second task to do the cancellation, if necessary. As there are two tasks involved, a scheduled thread pool of size 2 is created at (14). The first task is submitted at (15) to compute the factorial of a big number. A second task is scheduled at (16) to call the `cancel()` method on the `Future<BigInteger>` object representing the first task, after a delay of 10 milliseconds. If the first task has completed before the `cancel()` method is called, it has no effect on the first task. However, if the first task has not completed execution before the second task executes, the `cancel()` method will send an interrupt signal to the thread of the first task in an attempt to stop it. The output from the program shows that the first task was interrupted by the scheduled task. The `isCancelled()` method called on the `Future<BigInteger>` object at (18) confirms that the task was indeed cancelled. The output in this case depends on the execution time to compute the factorial for the specified number at (15) and the delay specified in the `schedule()` method at (16), before the call to the `cancel()` method is executed.

In summary, task cancellation is dependent on exploiting the thread interrupt system and providing an appropriate action in the interrupted thread executing the task.

Example 23.5 *Task Cancellation*

Click here to view code image

```
package executors;
import java.math.BigInteger;
import java.util.concurrent.*;

public class TaskCancellation {

  // Computation-intensive task:
  private static BigInteger bigFactorial(int number) {                    // (1)
    String threadName = Thread.currentThread().getName();
    BigInteger factorial = BigInteger.ONE;
    try {
      for (int i = 2; i <= number; i++) {
        factorial = factorial.multiply(BigInteger.valueOf(i));
        if (Thread.interrupted()) {                                       // (2)
          throw new InterruptedException();
        }
      }
      System.out.println(threadName + " completed.");
    } catch (InterruptedException e) {
      System.out.println(threadName + " interrupted.");                   // (3)
    }
    return factorial;
  }

  private static void timedTaskCancellation() {                           // (4)
    System.out.println("Timed task cancellation:");
    int number = 100000;
    long timeout = 1;
    TimeUnit unit = TimeUnit.SECONDS;
    System.out.println("NUMBER: " + number + ", "
                     + "TIMEOUT: " + timeout + " " + unit);
    ExecutorService exs = Executors.newSingleThreadExecutor();            // (5)
    Future<BigInteger> future = null;
    try {
      future = exs.submit(() -> bigFactorial(number));                    // (6)
      BigInteger result = future.get(timeout, unit);                      // (7)
      System.out.println("Factorial: " + result);                         // (8)
    } catch (TimeoutException e) {                                        // (9)
      System.out.println(e);
      System.out.println("cancel(true):  " + future.cancel(true));        // (10)
      System.out.println("isCancelled(): " + future.isCancelled());       // (11)
    } catch (InterruptedException | ExecutionException e) {
      System.out.println(e);                                              // (12)
    } finally {
      exs.shutdown();
    }
  }

  private static void scheduledTaskCancellation() {                       // (13)
    System.out.println("Scheduled task cancellation:");
```

```
    int number = 100000;
    long delay = 10;
    TimeUnit unit = TimeUnit.MILLISECONDS;
    System.out.println("NUMBER: " + number + ", DELAY: " + delay + " " + unit);
    ScheduledExecutorService ses = Executors.newScheduledThreadPool(2);   // (14)
    try {
      Future<BigInteger> future = ses.submit(() -> bigFactorial(number)); // (15)
      ses.schedule(() -> future.cancel(true), delay, unit);               // (16)
      TimeUnit.SECONDS.sleep(1);                                          // (17)
      System.out.println("isCancelled(): " + future.isCancelled());       // (18)
    } catch (InterruptedException e) {
      System.out.println(e);
    } finally {
      ses.shutdown();
    }
  }

  public static void main(String[] args) {
    timedTaskCancellation();
    System.out.println();
    scheduledTaskCancellation();
  }
}
```

Probable output from the program:

Click here to view code image

```
Timed task cancellation:
NUMBER: 100000, TIMEOUT: 1 SECONDS
java.util.concurrent.TimeoutException
pool-1-thread-1 interrupted.
cancel(true):  true
isCancelled(): true

Scheduled task cancellation:
NUMBER: 100000, DELAY: 10 MILLISECONDS
pool-2-thread-1 interrupted.
isCancelled(): true
```

23.3 The Fork/Join Framework

To harness the benefits of multicore architectures, the Fork/Join Framework in the `java.util.concurrent` package provides support for parallel programming that allows computations to run in parallel on multiple processors. In this section we provide an introduction to this framework which is also the basis of parallel streams (§16.9, p. 1009), but it is also worth exploring further in its own right.

The Fork/Join Framework is specially suited for problems that can be solved by the *divide-and-conquer* problem-solving paradigm. This technique is prominent in processing data in large arrays and collections—for example, in designing sorting and searching algorithms. The main step in a divide-and-conquer algorithm is to divide the task repeatedly into subtasks of the same type until the task size is smaller than a given threshold, such that a subtask can be solved directly. A simplified divide-and-conquer algorithm to solve a problem is presented below. When the task can be solved directly, this is called the *base case*. Otherwise, the algorithm proceeds with dividing the task into subtasks, solving each subtask recursively, and combining the results from each subtask, all the time the idea being that each subtask gets smaller and moves toward the base case. Solving a subtask recursively is referred to as *forking*, and combing the results of subtasks is referred to as *joining*, hence the name Fork/Join Framework.

Click here to view code image

```
if (taskSize < threshold)
  solve task directly                          // Base case
else {
  divide task into subtasks                    // Divide
  solve each subtask recursively               // Conquer
  combine results from each subtask            // Combine
}
```

Each subtask is solved by invocation of the divide-and-conquer algorithm recursively. The subtasks are candidates for *parallel processing*. The Fork/Join Framework facilitates both the management of threads and the use of available processors for parallel processing of subtasks.

Abstract classes shown in **Table 23.3** can be extended to define tasks. The abstract classes `RecursiveAction` and `RecursiveTask<V>` provide the abstract method `compute()` that defines the computation of the task. This method emulates the divide-and-conquer paradigm.

There are different ways to fork a subtask. One convenient way is to use the overloaded `invokeAll()` static method of the `ForkJoinTask<V>` class. This static method initiates the execution of the tasks and awaits their completion, returning any results, if necessary.

The overall execution of tasks is managed by the specialized executor service, `ForkJoinPool`, shown in **Figure 23.1**. It is used analogous to the executors encountered in the Executor Framework. The initial task can be submitted to the fork-join pool by calling the `invoke()` method that commences the performing of the task.

Table 23.3 *Task-Defining Classes*

Classes in the java.util.concurrent package to define tasks	Description
`ForkJoinTask<V>`	Abstract class that defines a task
`RecursiveAction extends ForkJoinTask<Void>`	Abstract class that should be extended to define a task that does not return a value
`RecursiveTask<V> extends ForkJoinTask<V>`	Abstract class that should be extended to define a task that returns a value

Example 23.6 illustrates using the Fork/Join Framework. It counts the number of values in an array of random integers that satisfy a given predicate—in this particular case, all numbers divisible by 7. The `FilterTask` class at (1) extends the `RecursiveTask<Integer>` abstract class to define a task for this purpose. It implements the `compute()` method at (2) that mimics the divide-and-conquer algorithm.

The array to filter for numbers divisible by 7 is successively divided into two equal subarrays given by the indices `fromIndex`, `toIndex`, and `middle`. If the size of a subarray is less than the threshold, the base case at (3) is executed. It constitutes iterating over the subarray to do the counting, and returning the result. Note that the base case does not entail further subdivision of the task.

If the size of the array is over the threshold, two new filter tasks are created at (4) with two equal subarrays from the current subarray. The two new filter tasks are forked at (5) by calling the `invokeAll()` static method. Recursion occurs when the two subtasks are executed and their `compute()` method is called. The result returned from each subtask can be obtained by calling the inherited `join()` method from the superclass `ForkJoinTask`. These results are combined at (6) to return the combined result of the current subarray.

The `main()` method of the `ForkJoinDemo` class shows the steps to set up the fork-join pool for execution. The initial task is created at (9). The fork-join pool is create at (10) and invoked on the task at (11) to initiate the computation. The `invoke()` method returns the combined result of all the subtasks that were created in executing the initial task.

The powerful paradigm of parallel streams is evident from the stream-based solution at (12) when compared with the solution using the Fork/Join Framework explicitly.

Example 23.6 *Filtering Values Using Fork/Join Framework*

Click here to view code image

```
package forkjoin;

import java.util.Random;
import java.util.concurrent.*;
import java.util.function.IntPredicate;
import java.util.stream.IntStream;

class FilterTask extends RecursiveTask<Integer> {                          // (1)
  public static final int THRESHOLD = 10_000;
  private int[] values;
  private int fromIndex;
  private int toIndex;
  private IntPredicate predicate;

  public FilterTask(int[] values, int fromIndex, int toIndex,
                    IntPredicate predicate) {
    this.values = values;
    this.fromIndex = fromIndex;
    this.toIndex = toIndex;
    this.predicate = predicate;
  }

  @Override
  protected Integer compute() {                                            // (2)
    if (toIndex - fromIndex < THRESHOLD) {
      // The base case:                                                     (3)
      var count = 0;
      for (int i = fromIndex; i < toIndex; i++) {
        if (predicate.test(values[i])) {
          ++count;
        }
      }
      return count;
    } else {
      // Create subtasks:                                                   (4)
      var middel = fromIndex + (toIndex - fromIndex) / 2;
      var subtask1 = new FilterTask(values, fromIndex, middel, predicate);
      var subtask2 = new FilterTask(values, middel, toIndex, predicate);
      // Fork and execute the subtasks. Await completion.                   (5)
      ForkJoinTask.invokeAll(subtask1, subtask2);
      // Combine the results returned by subtasks.                          (6)
      return subtask1.join() + subtask2.join();
    }
  }
}

public class ForkJoinDemo {
  public static void main(String[] args) {
    // Set up the array with the random int values:                         (7)
    final int SIZE = 1_000_000;
```

```
        int[] numbers = new Random().ints(SIZE).toArray();

        // Predicate to filter numbers divisible by 7:                          (8)
        IntPredicate predicate = i -> i % 7 == 0;

        // Create the initial task.                                             (9)
        var filterTask = new FilterTask(numbers, 0, numbers.length, predicate);
        // Create fork-join pool to manage execution of the task.               (10)
        var pool = new ForkJoinPool();
        // Perform the task, await completion, and return the result:           (11)
        var result = pool.invoke(filterTask);
        System.out.println("Fork/Join:       " + result);

        // Solution using parallel stream:                                      (12)
        System.out.println("Parallel stream: " +
                           IntStream.range(0, numbers.length).parallel()
                                    .map(i -> numbers[i]).filter(predicate).count());
    }
}
```

Probable output from the program:

```
Fork/Join:       142733
Parallel stream: 142733
```

23.4 Writing Thread-Safe Code

Thread-safety is a critical property of a concurrent application. Threads can generally execute their tasks concurrently without impending each other, unless they are sharing resources. Generally speaking, a shared resource can be any data that is visible to more than one thread. This could be an object or a primitive value, which multiple threads can access concurrently.

Each thread has its own execution stack that contains local variables of active methods during execution (§22.2, p. 1369). Any variable allocated on this stack is only visible to the thread that owns this stack, and is therefore automatically thread-safe. On the other hand, objects are placed in a heap which is shared by all threads. These objects are considered to be potentially visible to all threads. A thread can access any object in a heap if it has a reference to it. The challenge is to avoid thread interference (or race conditions) when concurrent threads are accessing a shared object.

As a running example, we will use different implementations of a counter to illustrate various approaches to achieving thread-safety. **Example 23.7** shows the interface `ICounter` at (1) that is implemented by a counter. The interface defines the two methods `increment()` and `getValue()` to increment and read the value in a counter, respectively.

The class `TestCounters` at (3) tests various counter implementations from (5) to (10) in the `main()` method at (4) by calling the method `runIncrementor()` defined at (11). This method creates a `Runnable` incrementor at (12) that calls the `increment()` method of a counter a fixed number of times (`NUM_OF_INCREMENTS`). The incrementor is submitted a fixed number of times to an executor service in the `try` block at (13), corresponding to the size of the thread pool of the executor service (`POOL_SIZE`). All together, a counter is incremented 10,000 times (`NUM_OF_INCREMENTS*POOL_SIZE`).

The output shown in **Example 23.7** is for the various counter implementations that will be discussed in this section. First up is an unsafe counter implementation at (2) which is not thread-safe, as we can see from the incorrect result in the output. The cause can be attributed to thread interference (§22.4, p. 1388) and memory consistency errors (§22.5, p. 1414) when accessing the shared counter field by the concurrent threads. We look at different solutions (in addition to the `synchronized` code) to implement mutually exclusive access of a shared object in a concurrent application.

Example 23.7 *Testing Counter Implementations*

Click here to view code image

```
package safe;
/** Interface that defines a basic counter. */
interface ICounter {                                                    // (1)
  void increment();
  int getValue();
}
```

Click here to view code image

```
package safe;
public class UnsafeCounter implements ICounter {                       // (2)
  private int counter = 0;

  @Override public int getValue()   { return counter; }
  @Override public void increment() { ++counter; }
}
```

Click here to view code image

```
package safe;
import java.util.concurrent.*;
import java.util.stream.IntStream;

public class TestCounters {                                             // (3)
```

```
  private static final int NUM_OF_INCREMENTS = 1000;
  private static final int POOL_SIZE = 10;

  public static void main(String[] args) throws InterruptedException {    // (4)

    UnsafeCounter usc = new UnsafeCounter();                              // (5)
    runIncrementor(usc);
    System.out.printf("Unsafe Counter: %24d%n", usc.getValue());

    VolatileCounter vc = new VolatileCounter();                           // (6)
    runIncrementor(vc);
    System.out.printf("Volatile Counter: %22d%n", vc.getValue());

    SynchronizedCounter sc = new SynchronizedCounter();                   // (7)
    runIncrementor(sc);
    System.out.printf("Synchronized Counter: %18d%n", sc.getValue());

    AtomicCounter ac = new AtomicCounter();                               // (8)
    runIncrementor(ac);
    System.out.printf("Atomic Counter: %24d%n", ac.getValue());

    ReentrantLockCounter rlc = new ReentrantLockCounter();                // (9)
    runIncrementor(rlc);
    System.out.printf("Reentrant Lock Counter: %16d%n", rlc.getValue());

    ReentrantRWLockCounter rwlc = new ReentrantRWLockCounter();           // (10)
    runIncrementor(rwlc);
    System.out.printf("Reentrant Read-Write Lock Counter: %d%n", rwlc.getValue());
  }

  public static void runIncrementor(ICounter counter) {                   // (11)
    // A Runnable incrementor to call the increment() method of the counter
    // a fixed number of times:
    Runnable incrementor = () -> {                                        // (12)
      IntStream.rangeClosed(1, NUM_OF_INCREMENTS).forEach(i->counter.increment());
    };

    // An executor service to manage a fixed number of incrementors:
    ExecutorService execService = Executors.newFixedThreadPool(POOL_SIZE);

    // Submit the incrementor to the executor service. Each thread executes
    // the same incrementor, and thereby increments the same counter.

    try {                                                                 // (13)
      IntStream.range(0, POOL_SIZE).forEach(i -> execService.submit(incrementor));
    } finally {
      execService.shutdown();
    }
    // Wait for all tasks to complete.
    try {
      while (!execService.awaitTermination(1, TimeUnit.SECONDS));
```

```
      } catch (InterruptedException e) {
        execService.shutdownNow();
      }
    }
  }
```

Probable output from the program:

Click here to view code image

```
Unsafe Counter:                    8495
Volatile Counter:                  8765
Synchronized Counter:             10000
Atomic Counter:                   10000
Reentrant Lock Counter:           10000
Reentrant Read-Write Lock Counter: 10000
```

Immutability

We have seen how to implement thread-safe access to shared resources in `synchronized` code that uses intrinsic locks (§22.4, p. 1387). However, if the shared resource is an immutable object, thread-safety comes for free.

An object is *immutable* if its state cannot be changed once it has been constructed. Since its state can only be read, there can be no thread interference and the state is always consistent.

Examples of immutable classes in the Java SE Platform API and designing immutable classes are discussed in §6.7, p. 356.

Volatile Fields

To optimize the execution of compiled code, the compiler performs optimizations on the bytecode. Typical optimizations are instruction reordering, method call inlining, and loop optimizations. For multithreaded applications, thread-specific optimizations can also be performed. It is quite common for threads to create their own private cached copies of variables (called *thread-local* variables) for performance reasons. Values of thread-local variables are synchronized with the values of their master variables in main memory at various times—for example, when a thread enters or exits synchronized code. Declaring a field as `volatile` informs the compiler that the field will be modified by different threads, and all reads and writes to the field should be reflected in the master copy in main memory. Without the `volatile` modifier, there is no guarantee that the latest value in a shared field will be visible to the different threads. The *visibility* of a shared `volatile` field is guaranteed by the following rule:

- *Volatile Field Rule:* A write to a `volatile` field *happens-before* every subsequent read of that field (§22.5, p. 1414).

The rule implies that a read on a `volatile` field will always see the value from the latest write on the `volatile` field. Without the `volatile` modifier, the threads may see different values in the shared field. The class `NonVolatileDemo` in **Example 23.8** illustrates the potential problem that can occur with visibility when different threads share data. The `boolean` field `stopThread` at (1) is used in the `run()` method at (3) of each child thread spawned in the loop at (2) to stop the thread when the field value becomes `true`. The main thread sets this value after sleeping for a little while. If the threads are using thread-local copies of the field `stopThread`, there is no guarantee when these threads will get to see the updated value, and therefore, might continue executing forever.

The class `VolatileDemo` in **Example 23.8** declares the field `stopThread` at (6) to be `volatile`. This is a typical use of the `volatile` modifier: to flag a condition to different threads. When the value is updated in the main thread, the child threads will see the updated value on the next read of the field, and are guaranteed to terminate.

Example 23.8 *Visibility of Shared Data*

Click here to view code image

```
package safe;
/** Potential problem with visibility of shared data. */
public class NonVolatileDemo {

  private static boolean stopThread = false;             // (1)

  public static void main(String[] args) throws InterruptedException {
    for (int i = 0; i < 2; i++) {                        // (2)
      new Thread(() -> {
        while (!stopThread) {                            // (3)
          System.out.println(Thread.currentThread().getName()
                             + ": Get me out of here!");
        }
      }, "T" + i).start();
    }
    Thread.sleep(1);                                     // (4)
    stopThread = true;                                   // (5)
  }
}
```

Click here to view code image

```
package safe;
/** Volatile field to guarantee visibility of shared data. */
```

```
public class VolatileDemo {

  private static volatile boolean stopThread = false;    // (6)

  // The main() method remains the same as in the NonVolatileDemo class.

}
```

Probable output from each of the programs:

```
...
T0: Get me out of here!
T0: Get me out of here!
T0: Get me out of here!
T1: Get me out of here!
...
```

All *reads* and *writes* are *atomic actions* for all `volatile` fields, regardless of whether they are object references or are of primitive data types. An *atomic action* is guaranteed to be performed by a thread without any interleaving—that is, without any race conditions. Such an action either performs in its entirety or not at all—akin to a database transaction. Most program actions are not actually atomic; even something as simple as an arithmetic operator takes more than one CPU cycle, and therefore can be interrupted before the action completes.

Note that atomicity holds only for read and write operations, not for non-atomic operations like the increment operator (`++`), that are actually executed in several steps. For example, the expression `++i` is equivalent to the following code:

```
int tmp = i;
tmp = tmp + 1;
i = tmp;
```

The expression `++i` actually is being evaluated as a read of the value in `i`, and then a write to `i` after the value has been incremented by `1`. If `i` was a shared `volatile` field, a different thread could read it between the atomic read and write operations, resulting in potential memory consistency errors, unless intermediate values in `i` did not matter. Computing the next value in `i` is dependent on the previous value in `i`. Declaring a variable with such data dependencies as `volatile` may not be adequate, as demonstrated by **Example 23.9** that implements the counter as a `volatile` field. Not surprisingly, the output from the program shows an incorrect result of incrementing the counter due to memory consistency errors.

The `volatile` keyword is enough to combat memory consistency problems. That is, it solves the visibility problem (writing to main memory), but not interleaving of operations on a shared variable (i.e., thread interference/race conditions). Solving both problems requires mutual exclusion. We explore solutions for implementing mutual exclusion other than `synchronized` code later in this section: atomic variables for mutual exclusion on shared single variables (**p. 1456**) and programmatic locks for mutual exclusion on shared resources (**p. 1460**).

Example 23.9 *Volatile Counter*

Click here to view code image

```
package safe;
public class VolatileCounter implements ICounter {
  private volatile int counter = 0;

  @Override public int getValue()   { return counter; }
  @Override public void increment() { ++counter; }
}
```

Probable output from the program in **Example 23.7**, **p. 1451**:

Click here to view code image

```
Volatile Counter:                  87672
```

There is no need to use a `volatile` field unless it is shared between threads, or if only atomic reads and writes are necessary on a field. A field of primitive type `long` or `double` does not guarantee atomic reads and writes, unless it is declared `volatile`. Also, a `final` field cannot be declared `volatile`, as it is immutable.

It is worth noting the differences between the `volatile` and the `synchronized` keywords. The keyword `volatile` is only applicable to fields, whereas the keyword `synchronized` is only applicable to a statement block or method. A `synchronized` statement cannot synchronize on `null`, but a `volatile` field may be `null`.

Since there are no locks involved when accessing a `volatile` field, there is no blocking of threads either. Performance overhead is also lightweight, compared to `synchronized` code which is bound by the whole regime of thread management, although the keyword signals to the compiler *not* to undertake certain optimizations. And while the `volatile` keyword cannot replace `synchronized` code in all situations, it is more efficient in certain situations where the visibility and atomicity of a shared field are overriding factors.

Atomic Variables

Unless the action on a shared `volatile` variable is atomic, thread-safety cannot be guaranteed because of potential race conditions. Non-atomic actions (such as the use of the increment operator) by different threads can result in interleaving of constituent actions like read, update, and write.

Shared lock-free thread-safe variables are implemented by the classes in the `java.util.concurrent.atomic` package (**Table 23.4**). The classes provide methods that implement *atomic actions*. The classes and the variables that denote their instances are called *atomic classes* and *atomic variables*, respectively.

Atomic variables also behave as if they are volatile, so there is no need to declare them explicitly with the `volatile` keyword—that is, the atomic actions of the atomic classes guarantee visibility of an atomic variable to other threads.

The atomic classes thus implement *non-blocking lock-free synchronization* of atomic variables by a predefined set of atomic actions, ensuring visibility and avoiding race conditions.

Table 23.4 *Selected Atomic Classes*

Atomic classes in the java.util.concurrent.atomic package	Mutable value that may be updated atomically
`AtomicBoolean`	A `boolean` value
`AtomicInteger implements Number`	An `int` value
`AtomicLong implements Number`	A `long` value
`AtomicReference<V>`	An object reference

Example 23.11 illustrates a thread-safe counter implemented using the `Atomic-Integer` class. The counter field is declared at (1) as an `AtomicInteger` and initialized to `0`. The `getValue()` and `increment()` methods at (2) and (3) delegate their operation to appropriate atomic methods of the `AtomicInteger` class. As the name implies, the `get()` method atomically returns the current value of the counter. The `increment-AndGet()` method atomically increments the counter and returns the new value (that is ignored in this example). The `incrementAndGet()` method entails more than one action, but none of them can be interrupted. The output shows that the counter was correctly incremented by the threads.

To understand how some of the atomic operations are implemented, we take a look at the `while` loop at (5), which implements the equivalent of the `incrementAndGet()` method at (4).

The strategy used in the `while` loop is known as *CAS: Compare and Set*, after the method called at (8). The `compareAndSet()` method will not set the counter to the new value unless the current value is equal to the expected value.

Assume that the current value in the counter is 100. Referring to the numbered lines in the code:

(6) The expected value is 100 (i.e., the current counter value).
(7) The new value is 101, after incrementing the expected value.
(8) The call to the `compareAndSet()` method in the conditional expression of the `if` statement returns `true`, since the current value (100) is equal to the expected value (100). The new value (101) is set in the counter—that is, it is correctly incremented by 1 from the previous value.
(9) The loop terminates.

Assume that by the time the current value is read again for comparison at (8) by the `compareAndSet()` method, it has been changed to 105 by another thread. This is possible by another thread in the time between the counter value is accessed at (6) by the `get()` method and then again at (8) by the `compareAndSet()` method—that is, there is a potential race condition.

(8) The call to the `compareAndSet()` method in the conditional expression of the `if` statement returns `false`, since the current value (105) is not equal to the expected value (100). The value in the counter is not changed as it could corrupt the state of the counter, and the loop continues until the counter value is correctly incremented by 1.

The `while` loop ensures that the value is always incremented by 1 from the previous value. Many of the methods in the atomic classes use a similar strategy to implement atomic operations.

It goes without saying that the `AtomicInteger` class should not be used as a replacement for the `Integer` class. We leave the inquisitive reader to explore the Atomic API further at leisure.

Example 23.10 *Atomic Counter*

Click here to view code image

```
package safe;
import java.util.concurrent.atomic.AtomicInteger;
```

```
public class AtomicCounter implements ICounter {
  private AtomicInteger counter = new AtomicInteger(0);                    // (1)

  @Override
  public int getValue() {                                                  // (2)
    return counter.get();
  }

  @Override
  public void increment() {                                                // (3)
    counter.incrementAndGet();                                             // (4)
//  while (true) {                                                         // (5)
//    int expectedValue = counter.get();                                   // (6)
//    int newValue = expectedValue + 1;                                    // (7)
//    if (counter.compareAndSet(expectedValue, newValue)) {// (8) Compare and Set.
//      return;                                                 // (9)
//    }
//  }
  }
}
```

Output from the program in **Example 23.7**, **p. 1451**:

Click here to view code image

```
Atomic Counter:                    10000
```

The `AtomicInteger` class provides the following constructors:

Click here to view code image

```
AtomicInteger()
AtomicInteger(int initialValue)
```

Create a new `AtomicInteger` with initial value 0 or with the given initial value, respectively.

Selected atomic methods from the `AtomicInteger` class are presented below.

```
int get()
```

Returns the current value.

```
void set(int newValue)
```

Sets the value to `newValue`.

```
int getAndSet(int newValue)
```

Atomically sets the value to `newValue` and returns the old value.

Click here to view code image

```
boolean compareAndSet(int expectedValue, int newValue)
```

Atomically sets the value to `newValue` if the current value == `expectedValue`, returning `true` if the operation was successful; otherwise, it returns `false`.

```
int addAndGet(int delta)
int getAndAdd(int delta)
```

Atomically adds the given value to the current value, returning the updated value or the previous value, respectively.

```
int decrementAndGet()
int getAndDecrement()
```

Atomically decrements the current value, returning the updated value or the previous value, respectively.

```
int incrementAndGet()
int getAndIncrement()
```

Atomically increments the current value, returning the updated value or the previous value, respectively.

Click here to view code image

```
int accumulateAndGet(int x, IntBinaryOperator accumulatorFunction)
int getAndAccumulate(int x, IntBinaryOperator accumulatorFunction)
```

Atomically updates the current value with the results of applying the given function to the current and given values, returning the updated value or the previous value, respectively.

Click here to view code image

```
int updateAndGet(IntUnaryOperator updateFunction)
int getAndUpdate(IntUnaryOperator updateFunction)
```

Atomically updates the current value with the results of applying the given function, returning the updated value or the previous value, respectively.

```
int    intValue()
double doubleValue()
float  floatValue()
long   longValue()
```

Returns the current value of this `AtomicInteger` as an `int`, `double`, `float`, and `long` after a widening primitive conversion, if necessary, respectively.

Intrinsic Locking Revisited

Intrinsic locking provides a blocking, lock-based, thread-safe solution for concurrent threads accessing mutable shared resources. The keyword `synchronized` is used to implement critical sections of code (synchronized methods/blocks) that rely on intrinsic locks to guarantee mutual exclusion to a single thread at a time (§22.4, p. 1387). Race conditions are thus avoided by mutual exclusion guaranteed by the critical section. The happens-before object lock rule (in this case for intrinsic locks) guarantees that any updates to the shared data done by one thread in a critical section will be visible to any other thread in another critical section guarded by the same intrinsic lock—thus eliminating memory consistency errors (§22.5, p. 1414).

Example 23.11 is an implementation of a thread-safe counter using `synchronized` methods for this particular case. Note that it is not necessary to declare the `counter` field as `volatile`, as both visibility and absence of race conditions are guaranteed by the synchronized code. The two operations on the counter are implemented as `synchronized` methods that are mutually exclusive, ensuring thread-safe use of the counter when accessed by different threads.

Example 23.11 *Synchronized Counter*

Click here to view code image

```
package safe;
public class SynchronizedCounter implements ICounter  {
  private int counter = 0;

  @Override public synchronized int getValue() { return counter; }
```

```
    @Override public synchronized void increment() { counter++; }
}
```

Output from the program in **Example 23.7**, **p. 1451**:

Click here to view code image

```
Synchronized Counter:              10000
```

Programmatic Locking

The Lock API in the `java.util.concurrent.locks` package provides flexible programmatic locking mechanisms that can be used to ensure thread-safety of shared mutable resources accessed by concurrent threads.

Intrinsic locking is based on the *acquire-release lock paradigm* to implement mutual exclusion. This paradigm is also the basis of locking mechanisms for mutual exclusion provided by the Lock API. This API also allows more control over *lock acquisition* (i.e., how the lock should be acquired) and *lock disciplines* (i.e., how to choose a thread among the waiting threads to acquire a released lock). Programmatic locking is thus more flexible, and can prevent threads from potential resource starvation (**§22.5**, **p. 1412**).

Selected interfaces and classes from the Lock API are shown in **Figure 23.3**. The main interfaces are `Lock` and `ReadWriteLock` that are implemented by the `ReentrantLock` and the `ReentrantReadWriteLock` classes, respectively. The `ReentrantReadWriteLock` class defines two nested classes, `ReadLock` and `WriteLock`, that implement the `Lock` interface. We explore the Lock API in the next section.

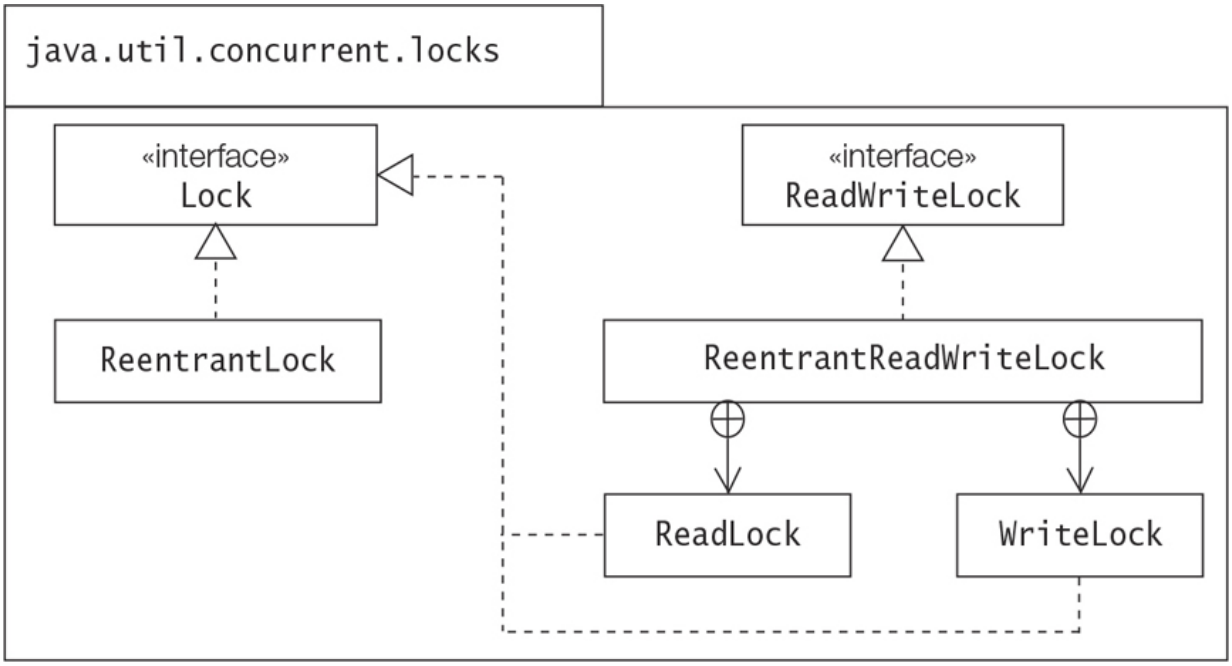

Figure 23.3 *Selected Interfaces and Classes in the Lock API*

Reentrant Lock

The `ReentrantLock` class implements the `Lock` interface. Instances of the `ReentrantLock` class are basically analogous to intrinsic locks. A thread needs to *lock* a `ReentrantLock` instance in order to gain exclusive access to a critical section guarded by the `ReentrantLock` instance. Afterward, the `ReentrantLock` instance must be *unlocked* so that other threads can try to gain access to the critical region.

The `ReentrantLock` class provides two constructors to create locks, with or without a *fairness policy*, as shown below at (1) and (2), respectively. The fairness policy is specified by the value `true` in the constructor. In this case, the *acquisition order* is that the longest-waiting thread for the lock is chosen to acquire the released lock. Otherwise, no guarantees are given as to how a thread will be chosen from the threads waiting to acquire the lock.

Click here to view code image

```
Lock frl = new ReentrantLock(true);  // (1) Fair reentrant lock
Lock rl = new ReentrantLock();       // (2) Reentrant lock with no fairness policy
```

```
ReentrantLock()
```

Creates an instance of `ReentrantLock`. This `ReentrantLock` instance does *not* use a fairness policy, and it is equivalent to `ReentrantLock(false)`.

```
ReentrantLock(boolean fair)
```

Creates an instance of `ReentrantLock` with the given fairness policy. The fair ordering policy is implied by the value `true`, allowing the longest-waiting thread to acquire the lock when it is released.

The basic idiom to use a reentrant lock for mutual exclusion is shown below. A thread first *locks* the `ReentrantLock` instance at (1) by calling the `lock()` method—we say that the thread *acquires* the lock. At (2), the thread *unlocks* the `ReentrantLock` instance by calling the `unlock()` method—we say that the thread *releases* the lock. The thread has exclusive access to the code executed between (1) and (2) if it acquires the lock. Using the `try-finally` statement ensures that the lock is always released in the `finally` block. In general, the `lock()` and `unlock()` methods can be called in separate methods.

Click here to view code image

```
frl.lock();               // (1) Acquire the lock.
try {
  // Exclusive access...
} finally {
  frl.unlock();           // (2) Release the lock.
}
```

The `ReentrantLock` class implements methods of the `Lock` interface that define different *lock acquisition policies*.

- *Unconditional locking*
  The `lock()` method implements this policy, where the thread must unconditionally wait to acquire the lock if it is not available immediately.
  Click here to view code image

  ```
  frl.lock();                                          // Unconditional locking.
  try { /* Lock acquired. Can access the resource. */ }
  finally { frl.unlock(); }
  ```

- *Polled locking*
  The `tryLock()` method implements this policy, where the call will return immediately with the value `false` if the lock is not available. The value `true` is returned if the lock is acquired.
  Click here to view code image

  ```
  if (frl.tryLock()) {                                  // Polled locking.
    try { /* Lock acquired. Can access the resource. */ }
    finally { frl.unlock(); }
  } else { /* Lock was not acquired. */ }
  ```

- *Timed locking*
  The `tryLock(timeout, timeunit)` method will also return `false` if the thread is not able to acquire the lock in the specified time. If the thread is interrupted while waiting for the lock, it will throw an `InterruptedException`.
  Click here to view code image

  ```
  try {
    if (frl.tryLock(100, TimeUnit.MILLISECONDS)) {          // Timed locking.
      try { /* Lock acquired. Can access the resource. */ }
      finally { frl.unlock(); }
    } else { /* Lock was not acquired. */ }
  } catch (InterruptedException iex) { iex.printStackTrace(); }
  ```

- *Interruptible locking*
  The `lockInterruptibly()` method throws an `InterruptedException` if the thread is interrupted while acquiring the lock or while waiting for the lock.
  Click here to view code image

```
try {
  frl.lockInterruptibly();                              // Interruptible locking.
  try { /* Lock acquired. Can access the resource. */ }
  finally { frl.unlock(); }
} catch (InterruptedException iex) { iex.printStackTrace(); }
```

More details are provided below about the lock acquisition policy implemented by the methods mentioned above. These methods are defined by the `Lock` interface and implemented by the `ReentrantLock` class.

```
void lock()
```

If the lock is not held by another thread, the *lock hold count* is set to 1 and the method returns immediately.

If the current thread already holds the lock, the lock hold count is incremented by 1 and the method returns immediately.

If the lock is held by another thread, the thread waits until the lock is acquired, at which point the lock hold count is set to 1.

```
void unlock()
```

If the current thread is the holder of this lock, the lock hold count is decremented. If the lock hold count now has the value `0`, the lock is released.

If the current thread is not the holder of this lock, then an `IllegalMonitorStateException` is thrown.

Note that a lock action on a lock must be matched by an unlock action.

The thread must make sure that an acquired lock is released after use, typically in the `finally` block of a `try-finally` statement.

```
boolean tryLock()
```

If the lock is not held by another thread, the *lock hold count* is set to 1 and the method returns `true`. This method does not honor fairness—it does not care if there are any threads already waiting for this lock.

If the current thread already holds the lock, the lock hold count is incremented by 1 and the method returns `true`.

If the lock is held by another thread, the method returns immediately with the value `false`.

Click here to view code image

```
boolean tryLock(long time, TimeUnit unit) throws InterruptedException
```

Acquires the lock if it is free within the given waiting time and the current thread has not been interrupted. In this case, the lock hold count is set to 1 and the value `true` is returned. The method respects the fair ordering policy if one has been specified for this lock.

If the current thread already holds the lock, the lock hold count is incremented by 1 and the method returns `true`.

If the current thread is interrupted while waiting to acquire the lock, an `InterruptedException` is thrown.

If timed out, the method returns the value `false`.

Click here to view code image

```
void lockInterruptibly() throws InterruptedException
```

If the lock is not held by another thread and the current thread is not interrupted, the *lock hold count* is set to 1 and the method returns immediately.

If the current thread already holds the lock, the lock hold count is incremented by 1 and the method returns immediately.

If the lock is held by another thread, the current thread waits for the lock. If the lock is acquired by the current thread while waiting, the lock hold count is set to 1.

If the current thread is interrupted while waiting to acquire the lock, an `InterruptedException` is thrown.

The following selected methods for querying a lock are only defined by the `ReentrantLock` class:

```
boolean isFair()
```

Returns `true` if this lock has the fairness policy set to `true`.

```
boolean isLocked()
```

If the lock is held by any thread, this method returns `true`; otherwise, it returns `false`.

```
int getQueueLength()
```

Returns an estimate of the number of threads waiting to acquire this lock.

```
boolean hasQueuedThreads()
```

Returns `true` if any threads are waiting to acquire this lock; otherwise, it returns `false`.

Click here to view code image

```
boolean hasQueuedThread(Thread thread)
```

Returns `true` if the given thread is waiting to acquire this lock; otherwise, it returns `false`.

Example 23.12 illustrates using a reentrant lock to implement a thread-safe counter. The class `ReentrantLockCounter` at (1) implements the `ICounter` interface (**Example 23.7**). It defines a counter whose value can be read by the method `get-Value()` at (1), and incremented by the method `increment()` at (5). The class instantiates a `ReentrantLock` instance at (2) that is used in the two methods to implement exclusive access to the `counter` field.

The `getValue()` method at (4) acquires the `rl` lock, reads and stores the current `counter` value locally, unlocks the `rl` lock, and returns the locally stored `counter` value. The `increment()` method at (5) acquires the `rl` lock, increments the `counter`, calls the `getValue()` method (ignoring the returned value), and releases the `rl` lock. The setup with the locks ensures that any read or write operation will have exclusive access to the

`counter` field. When testing the `ReentrantLockCounter` class with **Example 23.7, p. 1451**, the output shows that the counter was incremented correctly by the threads.

Note that the expression of a `return` statement in a `try` block is evaluated first and its value stored locally, before the `finally` block is executed. The stored result is returned after the completion of the `finally` block.

Intrinsic locks are *reentrant*: A thread that has acquired an intrinsic lock can acquire the same intrinsic lock again immediately (**§22.4, p. 1390**). This is also true of the locks implemented by the `ReentrantLock` class. In other words, explicit locks implemented by the `ReentrantLock` class are reentrant, as their name implies. The following code relies on the lock being reentrant:

Click here to view code image

```
frl.lock();
frl.lock();
// Access the resource. Lock hold count is 2.
frl.unlock();
frl.unlock();
```

Example 23.12 also illustrates the reentrant nature of explicit locks. The `increment()` method of the `ReentrantLockCounter` class calls the method `getValue()`. The current thread already holds the `rl` lock when the `getValue()` method is called. The `getValue()` method also calls the `lock()` method on the `rl` lock. If `rl` was not reentrant, the current thread would not be able to proceed in the `getValue()` method and would starve waiting for the lock to become available, as it already has been acquired by the current thread. The reentrant nature of the `rl` lock allows the current thread to reacquire the lock and execute the `getValue()` method.

Example 23.12 *Reentrant Lock Counter*

Click here to view code image

```
package safe;
import java.util.concurrent.locks.*;

public class ReentrantLockCounter implements ICounter {                // (1)

  private Lock rl = new ReentrantLock();                                // (2)
  private int counter = 0;                                              // (3)

  @Override
  public int getValue() {                                               // (4)
    rl.lock();
    try {
```

```
        return counter;
      } finally { rl.unlock(); }
    }

    @Override
    public void increment() {                                          // (5)
      rl.lock();
      try {
        counter++;                                                     // (6)
        getValue();                                                    // (7)
      } finally { rl.unlock(); }
    }
  }
```

Output from the program in **Example 23.7, p. 1451**:

Click here to view code image

```
Reentrant Lock Counter:            10000
```

Reentrant Read-Write Lock

The explicit lock implemented by the `ReentrantLock` class is *mutually exclusive*—that is, only one thread at a time can obtain the lock and thereby access the shared data. There are cases where shared data is read more often than it is modified—that is, there are more *readers* (or *reader threads*) than *writers* (or *writer threads*) accessing the shared data. Taking into consideration such behavior to improve the level of concurrency would be a challenge using a mutually exclusive lock, as the lock acquisition is exclusive, irrespective of whether it is a read or a write operation.

The `ReadWriteLock` interface provides a more flexible and sophisticated locking mechanism than a mutually exclusive lock for accessing shared data. **Figure 23.3** shows that the `ReentrantReadWriteLock` class implements the `ReadWriteLock` interface. It implements a *read-write lock* that actually maintains *a pair of associated locks*, one for read operations and one for write operations. The *read lock* is an instance of the `ReadLock` inner class and the *write lock* is an instance of the `WriteLock` inner class, both of which implement the `Lock` interface. The `ReadWriteLock` interface defines the methods to obtain the read and the write lock of a read-write lock.

The following methods are defined in the `ReadWriteLock` interface:

```
Lock readLock()
```

```
Lock writeLock()
```

These methods are implemented by inner classes in the `ReentrantReadWriteLock` class:

Click here to view code image

```
ReentrantReadWriteLock.ReadLock readLock()
ReentrantReadWriteLock.WriteLock writeLock()
```

Return the lock used for reading and for writing, respectively. The inner classes `ReadLock` and `WriteLock` implement the `Lock` interface.

The `ReentrantReadWriteLock` class provides the following constructors to create read-write locks with or without a fairness policy:

```
ReentrantReadWriteLock()
```

Creates an instance of `ReentrantLock` with non-fair ordering policy—imposing no ordering for lock access. A non-fair lock may be acquired by a thread at the expense of other waiting reader and writer threads.

Click here to view code image

```
ReentrantReadWriteLock(boolean fair)
```

Creates an instance of `ReentrantLock` with the given fairness policy. The value `true` indicates to use a fair ordering policy.

The basic idea behind a read-write lock is that a thread can use its read and write locks to perform thread-safe read and write operations on shared data, respectively. A thread is bound by the following lock acquisition rules for the read lock and the write lock of a read-write lock:

- *The read lock*: Only if no thread holds *the write lock* can a thread acquire the read lock. This means that the read lock can be shared by several reader threads— that is, a group of reader threads can access the shared data concurrently, as long as the write lock is not held by any thread.

- *The write lock*: Only if no thread holds *the write lock or the read lock* can a thread acquire the write lock. The write lock is thus mutually exclusive for writer threads—only one writer thread at a time can access the shared data.

Since the inner classes `ReadLock` and `WriteLock` implement the `Lock` interface, the lock acquisition policies of the `Lock` interface discussed earlier (p. 1462) are applicable to the read lock and the write lock, bearing in mind the specialized lock acquisition rules mentioned above for the read-write lock:

- *Unconditional locking* using the blocking `lock()` method
- *Polled locking* using the non-blocking `tryLock()` method
- *Timed locking* using the `tryLock(timeout, timeunit)` method
- *Interruptible locking* using the `lockInterruptibly()` method

As with any implementation of the `Lock` interface, the read lock and the write lock of a read-write lock must be released when done:

- *Released* using the `unlock()` method

Example 23.13 *Reentrant Read-Write Lock Counter*

Click here to view code image

```
package safe;
import java.util.OptionalInt;
import java.util.concurrent.locks.*;

public class ReentrantRWLockCounter implements ICounter {

  private ReadWriteLock rwl = new ReentrantReadWriteLock();           // (1)
  private Lock readLock = rwl.readLock();                             // (2)
  private Lock writeLock = rwl.writeLock();                           // (3)
  private int counter = 0;

  @Override
  public int getValue() {                                             // (4)
    readLock.lock();
    try {
//      System.out.println(Thread.currentThread().getName() + ": " + counter);
      return counter;
    } finally { readLock.unlock(); }
  }

  @Override
  public void increment() {                                           // (5)
    writeLock.lock();
    try {
```

```
      counter++;
    } finally { writeLock.unlock(); }
  }

  public int incrementAndGet() {                                    // (6)
    writeLock.lock();                // Acquire write lock.
    try {
      return ++counter;                                             // (7)
//    ++counter;                     // Increment.                  // (8)
//    return getValue();             // Get the new value.          // (9)
    } finally { writeLock.unlock(); } // Release write lock.
  }

  public int getAndIncrement() {                                    // (10)
    writeLock.lock();                // Acquire write lock.
    try {
      return counter++;              // Get and increment.          // (11)
    } finally { writeLock.unlock(); } // Release write lock.
  }

  public boolean incrIfPossible() {                                 // (12)
    if (writeLock.tryLock()) {          // Attempts to acquire the write lock.
      try {                             // Write lock acquired.
        counter++;
        return true;
      } finally { writeLock.unlock(); } // Write lock released.
    } else {                            // Write lock not acquired.
      return false;
    }
  }

  public OptionalInt getIfPossible() {                              // (13)
    if (readLock.tryLock()) {           // Attempts to acquire the read lock.
      try { return OptionalInt.of(counter); }
      finally { readLock.unlock(); }    // Read lock released.
    } else {                            // Read lock not acquired.
      return OptionalInt.empty();
    }
  }
}
```

Output from the program in **Example 23.7, p. 1451**:

Click here to view code image

```
Reentrant Read-Write Lock Counter: 10000
```

In **Example 23.13**, the class `ReentrantRWLockCounter` implements a thread-safe counter using a `ReentrantReadWriteLock`. It is a reworking of the counter in **Example 23.12** that uses a `ReentrantLock`. In **Example 23.13**, the write lock allows mutually exclusive access to write operations and the read lock is shared concurrently by read operations. The numbered comments below correspond to the numbered code lines in **Example 23.13**.

(1)–(3) A non-fair `ReentrantReadWriteLock` is created, and its read and write locks obtained.
(4) The read lock acquired by a thread in the `getValue()` method ensures that the counter can be read safely, as no other thread can hold the write lock to modify the counter. This approach allows concurrent read operations and prevents concurrent write operations on the mutable counter, making it thread-safe. The read lock is guaranteed to be released in the `finally` block of the `try-finally` statement.
(5) The `increment()` method uses the write lock to provide exclusive access to increment the counter.
(6) The `incrementAndGet()` method shows that it is safe to increment *and* read the counter when the thread holds the write lock, as no other thread can access the counter. The `return` statement at (7) is equivalent to the statements at (8) and (9). The call to the `getValue()` method at (9) acquires the read lock to read the counter value, while the current write thread is still holding the write lock. This illustrates the reentrant nature of the read-write lock, which allows a thread holding the write lock to acquire the read lock.
(10) The `getAndIncrement()` method shows that it is safe to read *and* increment the counter when the thread holds the write lock, as no other thread can access the counter.
(12) The `incrIfPossible()` method illustrates polling for the write lock. It uses the non-blocking `tryLock()` method in an attempt to acquire the write lock. If successful, it increments the counter, releases the write lock, and returns `true`. Otherwise, it returns `false`.
(13) The `getIfPossible()` method illustrates polling for the read lock. It uses the non-blocking `tryLock()` method in an attempt to acquire the read lock. If successful, it releases the read lock, and returns an `OptionalInt` with the value of the counter. Otherwise, it returns an empty `OptionalInt`, since the read lock could not be acquired.

Note that the methods in **Example 23.13** implement *atomic actions*. When testing the `ReentrantRWLockCounter` class with **Example 23.7**, **p. 1451**, the output shows that the counter was incremented correctly by the threads.

The `ReentrantReadWriteLock` class also provides miscellaneous methods that can be used to query a read-write lock:

```
boolean isFair()
```

Determines whether this lock has its fairness policy set to `true`.

```
int getReadHoldCount()
int getWriteHoldCount()
```

Queries the number of reentrant read holds and reentrant write holds on this lock by the current thread, respectively.

Click here to view code image

```
boolean isWriteLockedByCurrentThread()
```

If the write lock is held by the current thread, returns `true`; otherwise, it returns `false`.

The fairness policy comes into play when the currently held lock is released and there are threads waiting for a lock. For a read-write lock, there may be both writer threads and reader threads waiting to acquire a lock. The following aspects of a read-write lock's fairness policy should be noted:

- In order to assign the read lock, the fairness policy basically chooses between *one longest-waiting writer thread* among any waiting writer threads to assign the write lock and *the longest-waiting group of reader threads* among any waiting reader threads. Whichever of these has waited the longest is chosen.
- A thread can only acquire a fair read lock if both the write lock is free and there are no waiting writer threads. The thread will only acquire the read lock after the oldest currently waiting writer thread has acquired and released the write lock.
- A thread can only acquire a fair write lock if both the read lock and write lock are free; otherwise, it will block.

Analogous to the `tryLock()` methods of the `ReentrantLock` class, the `tryLock()` methods of the inner classes `ReadLock` and `WriteLock` do not honor the fairness policy, acquiring the lock immediately if available, regardless of any waiting threads.

Analogous to the reentrant nature of a `ReentrantLock`, the read-write lock allows both reader and writer threads to reacquire its read lock or write lock. Note that a thread that holds the write lock can acquire the read lock, but not vice versa. A reader thread that tries to acquire the write lock will block and never succeed.

Although a `ReentrantReadWriteLock` allows a greater level of concurrency to access shared data than a `ReentrantLock`, many factors can influence the performance of a read-write lock. Profiling is recommended to gauge the impact of such factors as its premises of more frequent reads than writes, duration of read and write operations, and number of cores available to leverage parallel execution. However, other thread-safe solutions provided by the Concurrency API for accessing shared data should also be considered.

23.5 Special-Purpose Synchronizers

The `java.util.concurent` package provides classes that implement useful special-purpose synchronization idioms that impose specific behavior on how threads can collectively make progress through some algorithm. We consider two such classes in this section: `CyclicBarrier` and `CountDownLatch`.

Cyclic Barrier

The `java.util.concurrent.CyclicBarrier` class implements a *barrier* upon which a set of threads have to synchronize before they are all allowed to proceed. The threads may need to wait for one another until all threads reach the barrier. When this happens the barrier will *trip*, allowing them to continue. The barrier is *cyclic*, as the same barrier can be reused after the waiting threads have been released.

The following constructors of the `CyclicBarrier` class can be used to create a barrier.

Click here to view code image

```
CyclicBarrier(int parties)
CyclicBarrier(int parties, Runnable barrierAction)
```

Both methods create a barrier that will trip when the specified number of `parties` (i.e., threads) are waiting for it—that is, when these invoke the `await()` method on it.

The barrier created by the second method will execute the given *barrier action* when the barrier is tripped. This action is performed by the *last* thread arriving at the barrier.

The declaration at (1) below creates a barrier that will trip when three threads synchronize on it. The declaration at (2) creates a barrier that will trip when five threads are waiting for it. In addition, a *barrier action* is defined that will be performed when the barrier is tripped. Typically, a barrier action can be used for consolidating any results computed by the threads thus far.

Click here to view code image

```
CyclicBarrier barrier1 = new CyclicBarrier(3);                    // (1)
CyclicBarrier barrier2 = new CyclicBarrier(5, () ->
         System.out.println("All results accumulated."));          // (2)
```

A cyclic barrier is *shared* by the threads wishing to synchronize on it. When a thread wants to wait at the barrier, it calls the `await()` method on the barrier. This defines a *barrier point*. The call blocks at the barrier point until the number of threads required to trip the barrier is reached—that is, the remaining threads also invoke the `await()` method. Otherwise, the barrier will not trip as long as the number of waiting threads is less than the number required by the barrier.

Click here to view code image

```
// Thread t1 waits on barrier1 at this barrier point.
barrier1.await();            // Can continue when the barrier is tripped.
...
// Thread t2 waits on barrier1 at this barrier point.
barrier1.await();            // Can continue when the barrier is tripped.
...
// Thread t3 waits on barrier1 at this barrier point.
barrier1.await();            // Can continue when the barrier is tripped.
...
```

The barrier `barrier1` is tripped when any three threads invoke the `await()` method on this barrier—for example, threads `t1`, `t2`, and `t3`, as shown above. When tripped, the waiting threads can proceed, and the cyclic barrier resets itself and can be tripped again.

The `CyclicBarrier` class also defines methods to query and reset a barrier.

Click here to view code image

```
int await() throws InterruptedException, BrokenBarrierException
int await(long timeout, TimeUnit unit) throws InterruptedException,
                                              BrokenBarrierException,
                                              TimeoutException
```

Both methods wait until all parties have invoked the `await()` method on this barrier, or until the specified waiting time elapses, as in the second method.

These methods return the *arrival index* of the current thread, index *N* `-1` for the first thread to arrive and `0` for the last thread to arrive, where *N* is the number of parties required to trip this barrier.

An `InterruptedException` is thrown if the current thread was interrupted while waiting.

A `BrokenBarrierException` is thrown if another thread was interrupted or timed out while the current thread was waiting, or the barrier was reset, or the barrier was broken when the `await()` method was called, or any barrier action failed due to an exception.

A `TimeoutException` is thrown if any specified timeout elapses. The barrier is then broken.

```
int getParties()
```

Returns the number of parties required to trip this barrier.

```
boolean isBroken()
```

Queries whether this barrier is in a *broken state*. A barrier can break due to an interruption, a timeout, the last reset, or a failed barrier action due to an exception. It returns `true` if any of the parties broke the barrier; otherwise, it returns `false`.

```
void reset()
```

Resets the barrier to its initial state—that is, the wait count of the barrier is set to 0. Any parties currently waiting at this barrier will return with a `Broken-BarrierException`.

Example 23.14 illustrates using a single cyclic barrier that is shared by three threads. The barrier is declared at (2), requiring three parties and having the barrier action at (1) being performed when the barrier is tripped. The task declared at (3) calls the `await()` method on the barrier when it wants to synchronize on the barrier. Note the checked exceptions that can be thrown by the `await()` method. The threads are created and started at (4).

The output from **Example 23.14** shows that each thread waited at the barrier point and announced it was released after the barrier tripped. The barrier action is performed by thread `T2` that was the last one to arrive at the barrier point.

Example 23.14 *Cyclic Barrier*

Click here to view code image

```
package synchronizers;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierDemo {
  public static final int PARTIES = 3;
```

```
  public static void main(String args[]) {
    Runnable barrierAction =  () ->                                          // (1)
          System.out.println("Barrier action by "
                             + Thread.currentThread().getName()
                             + ": All tasks are released.");

    CyclicBarrier barrier = new CyclicBarrier(PARTIES, barrierAction);     // (2)

    Runnable task = () -> {                                                  // (3)
      String threadName = Thread.currentThread().getName();
      try {
        System.out.println(threadName + " is now waiting");
        barrier.await();                                          // Barrier point.
        System.out.println(threadName + " is now released");
      } catch (BrokenBarrierException | InterruptedException e) {
        e.printStackTrace();
      }
    };

    for (int i = 0; i < PARTIES; i++) {                                      // (4)
      new Thread(task, "T" + (i+1)).start();
    }
  }
}
```

Probable output from the program:

Click here to view code image

```
T1 is now waiting
T3 is now waiting
T2 is now waiting
Barrier action by T2: All tasks are released.
T2 is now released
T1 is now released
T3 is now released
```

Threads can of course share more than one barrier, having a different number of parties and different barrier actions.

Once a barrier is tripped, its thread wait count is reset to zero, and the barrier can be used again to implement a barrier point by calling the `await()` method.

If more threads than the required number invoke the `await()` method, then the required number of threads will cause the barrier to trip. This will allow the required number to

continue and will reset the wait count, and the extra threads will have to wait for the barrier to trip again.

The cyclic barrier implements the *all-or-none model*. Either all waiting threads successfully continue execution past the `await()` method, or none do. If one thread waiting at a barrier point leaves prematurely—which could be due to an interruption, timeout, or failure—then all other waiting threads also leave abnormally via the checked `java.util.concurrent.BrokenBarrierException`. In this case, the barrier is said to be *broken*. The method `isBroken()` can be used to determine whether this is the case. A broken barrier can be reset and reused.

Count-Down Latch

The `java.util.concurrent.CountDownLatch` class implements a *count-down latch* that allows one or more threads to wait until the latch goes up, and then the waiting threads are allowed to proceed. The latch goes up depending on one or more other threads performing a specific number of countdown operations.

The `CountDownLatch` class provides the following constructor to create a count-down latch.

```
CountDownLatch(int count)
```

Creates a `CountDownLatch` initialized with the given `count`. The `count` specifies the number of times the `countDown()` method must be invoked to raise the latch to allow threads waiting on the `await()` method to pass through.

A count-down latch is created with an initial count. One or more threads can call the `await()` method of the latch to wait until the count reaches zero. The count is decremented by calls to the `countDown()` method of the latch by one or more other threads. Note that it is the number of calls to the `countDown()` method that raises the latch, independent of the number of threads involved. The `countDown()` method is used in tandem with the `await()` method to operate the latch. The latch can only go up once, and therefore cannot be reused.

Click here to view code image

```
void await() throws InterruptedException
boolean await(long timeout, TimeUnit unit) throws InterruptedException
```

Both methods cause the current thread to wait until one of the following events occurs: The latch count reaches 0, the thread is interrupted, or if any specified waiting time has elapsed.

In the first method, it is possible that the current thread may wait forever, if the count never reaches 0.

The second method returns `true` if the count reached 0 and `false` if the waiting time elapsed before the count reached 0.

```
void countDown()
```

Decrements the current count of the latch if it is greater than 0, releasing all waiting threads if the new count reaches 0. Nothing happens if the current count is already 0.

Given that the latch below is shared by threads `t1`, `t2`, and `t3`:

Click here to view code image

```
CountDownLatch latch = new CountDownLatch(3);    // (1) Latch count is 3.
```

Threads `t1` and `t2` call the `await()` method and wait at (2) and (3), respectively, for the latch to be raised:

Click here to view code image

```
// Thread t1:
latch.await();           // (2) Waiting for latch to be raised.
...
// Thread t2:
latch.await();           // (3) Waiting for latch to be raised.
...
```

Thread `t3` calls the `countDown()` method on the latch, decrementing the count each time. Note that the thread calling the `countDown()` method does *not* wait for the count to reach 0. The count reaches 0 when the call at (4) is executed, upon which the latch is raised, and waiting threads `t1` and `t2` can continue execution past the `await()` method call.

Click here to view code image

```
// Thread t3:
latch.countDown();
```

```
...
latch.countDown(),
...
latch.countDown();         // (4) Count reaches 0. Latch released at (2) and (3).
...
```

Example 23.15 illustrates a typical scenario where an administrator thread (in this case, the main thread) submits tasks (defined by the `Task` class at (7)) to individual threads and uses two count-down latches to start and wait for the tasks to finish.

The two latches, `startLine` and `finishLine`, are declared at (1) and (2), respectively. The `startLine` latch is an *on/off latch* with a count of 1. All submitted tasks invoke the `await()` method at (9) to wait until the `startLine` latch is raised by the main thread by invoking the `countDown()` method at (5) to decrement the count to 0.

The `finishLine` latch is a latch with a count of *N*. It is the converse of the `startLine` latch. All submitted tasks count down the `finishLine` latch by invoking the `countDown()` method at (10) and continue execution. The main method invokes the `await()` on the `finishLine` latch and waits until the latch is raised by all submitted tasks, decrementing the count to 0.

Example 23.15 *Count-Down Latch*

Click here to view code image

```
package synchronizers;
import static java.lang.System.out;
import java.util.concurrent.*;

public class CountDownLatchTest {
  public static final int N = 3;

  public static void main(String[] args) throws InterruptedException {
    CountDownLatch startLine = new CountDownLatch(1);              // (1)

    CountDownLatch finishLine = new CountDownLatch(N);             // (2)

    ExecutorService es = Executors.newFixedThreadPool(N);          // (3)
    String threadName = Thread.currentThread().getName();
    try {
      for (int i = 0; i < N; ++i) {    // (4) Submit tasks.
        es.submit(new Task(startLine, finishLine));
      }
      out.println(threadName + ": Let all tasks proceed.");
      startLine.countDown();           // (5) Count down to let all tasks proceed.
      finishLine.await();              // (6) Wait for all tasks to finish.
      out.println(threadName + ": All tasks done.");
```

```
    } finally {
      es.shutdown();
    }
  }
}

class Task implements Runnable {                                    // (7)
  private final CountDownLatch startLine;
  private final CountDownLatch finishLine;

  public Task(CountDownLatch start, CountDownLatch finish) {
    this.startLine = start;
    this.finishLine = finish;
  }

  @Override
  public void run() {                                                // (8)
    String threadName = Thread.currentThread().getName();
    try {
      out.println(threadName + ": Waiting to proceed.");
      startLine.await();               // (9) Wait to proceed.
      out.println(threadName + ": Running ... ");
      finishLine.countDown();          // (10) Count down the latch & continue.
      out.println(threadName + ": Latch count decremented.");
    } catch (InterruptedException ex) {
      ex.printStackTrace();
    }
  }
}
```

Probable output from the program:

Click here to view code image

```
main: Let all tasks proceed.
pool-1-thread-3: Waiting to proceed.
pool-1-thread-2: Waiting to proceed.
pool-1-thread-1: Waiting to proceed.
pool-1-thread-1: Running ...
pool-1-thread-2: Running ...
pool-1-thread-2: Latch count decremented.
pool-1-thread-3: Running ...
pool-1-thread-1: Latch count decremented.
pool-1-thread-3: Latch count decremented.
main: All tasks done.
```

23.6 Synchronized Collections and Maps

The primary goal of the Collections Framework is to provide support for efficient computation on collections that can be used in single-threaded applications. In the rest of this chapter we explore the support provided by the Collections Framework and the Concurrency Framework to create thread-safe collections that can be used in multi-threaded applications. We start with *synchronized collections* provided by the `java.util` package, and explore *concurrent collections* found in the `java.util.concurrent` package in the next section (p. 1482).

By a *thread-safe collection* we mean a collection whose elements can be processed by concurrent threads without ever being in an inconsistent state.

Unmodifiable collections cannot be changed structurally (add and remove operations are prohibited) and their elements cannot be replaced (throw an `UnsupportedOperationException`), making the unmodifiable collection immutable (§12.2, p. 649). This guarantees that an unmodifiable collection is thread-safe, and multiple reader threads can safely access the elements concurrently as long as it is done via the unmodifiable collection.

Click here to view code image

```
// Thread-safe unmodifiable list:
List<String> list = List.of("Tom", "Dick", "Harriet");
list.add("Harry");                          // UnsupportedOperationException
list.set(0, "Tommy");                       // UnsupportedOperationException
list.remove("Dick");                        // UnsupportedOperationException
System.out.println(list.get(0));            // "Tom"
```

Unmodifiable views of collections are backed by an underlying collection, where changes in the underlying collection are reflected in the view (§15.11, p. 856). Such a view also cannot be modified, and only query methods are passed to the underlying collection. The view can be considered effectively immutable, and thus thread-safe, if the backing collection is effectively immutable, or if the only reference to the backing collection is through the unmodifiable view. An example of such an unmodifiable view is given below:

Click here to view code image

```
// Effectively immutable list view:
List<Integer> list = new ArrayList<>();
list.add(2021); list.add(2022);
List<Integer> immutablelist = Collections.unmodifiableList(list);
immutablelist.add(2023);                    // UnsupportedOperationException
immutablelist.set(0, 2023);                 // UnsupportedOperationException
```

```
immutablelist.remove(2021);                    // UnsupportedOperationException
System.out.println(immutablelist.get(0));      // 2021
```

It is worth keeping in mind that a thread-safe collection does not imply that its elements are thread-safe. It might be necessary to employ an appropriate synchronization mechanism on the elements in order to modify them in a thread-safe way.

The `java.util.Collections` class provides methods to create *synchronized collections* that are thread-safe. Actually, these collections are *synchronized views of collections* as they are backed by an *underlying collection*. The methods provided accept a collection and return a view of this collection in which the getter and setter methods use synchronized blocks to delegate operations to the underlying collection—that is, the methods provide a synchronized wrapper around the underlying collection. A single intrinsic lock on the synchronized collection implements mutual exclusion of operations, but this can potentially become a point of contention among threads. Synchronized collections also incur the performance penalty associated with intrinsic locking. Other thread-safe solutions explored in this chapter should be considered before settling on using a synchronized view to make a collection thread-safe.

Click here to view code image

```
static <E> Collection<E>   synchronizedCollection(Collection<E> c)
static <E> List<E>         synchronizedList(List<E> list)
static <E> Set<E>          synchronizedSet(Set<E> set)
static <E> SortedSet<E>    synchronizedSortedSet(SortedSet<E> set)
static <E> NavigableSet<E> synchronizedNavigableSet(NavigableSet<E> set)
```

Return a synchronized (thread-safe) *view* collection that has the same type as the specified backing collection. It is imperative that any access to the backing collection is accomplished through the returned collection (or its views) to guarantee serial access.

Analogous to an unmodifiable view collection (**§15.11**, **p. 856**), the synchronized collection returned by the `synchronizedCollection()` method does not delegate the `equals()` and the `hashCode()` methods to the backing collection. Instead, the returned collection uses the corresponding methods inherited from the `Object` class. This is to safeguard the contract of these methods when the backing collection is a set or a list. However, the synchronized collections returned by the other methods above do not exhibit this behavior.

Click here to view code image

```
static <K,V> Map<K,V>       synchronizedMap(Map<K,V> map)
static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> map)
static <K,V> NavigableMap<K,V>
                            synchronizedNavigableMap(NavigableMap<K,V> map)
```

Return a synchronized (thread-safe) *view* map that has the same type as the backing map that is specified. It is imperative that any access to the backing map is accomplished through the synchronized map (e.g., its key, entry, or values views) to guarantee thread-safe serial access.

We use the synchronized list as an example in this section. For using the other synchronized collections and maps, familiarity with their unsafe counterparts will go a long way. For the most part, the synchronized collections provide mutually exclusive operations corresponding to the operations on their unsafe counterparts.

Serial Access

In order to guarantee thread-safety, *all* access to elements of the underlying collection should be through the synchronized collection. Although getter and setter methods are synchronized in a synchronized collection, this is *not* the case for methods that implement *serial access* (e.g., iteration using an iterator or a stream).

As multiple methods can be called for an operation on a synchronized collection when using the iterator, we need to ensure that these are executed as a single mutually exclusive operation. Iteration thus requires a coarse-grained manual synchronization on the synchronized collection for deterministic results. A more general case of coarse-grained synchronization is discussed in the next subsection (**p. 1480**).

Example 23.16 illustrates the idiom used for serial access over a synchronized collection. A synchronized list is created at (1) and populated. A `Runnable` is implemented at (2) to remove a certain year from the synchronized list.

Serial access over the synchronized list is attempted using an explicit iterator. However, serial access operations are all done in a synchronized block requiring the intrinsic lock on the synchronized list, as at (3). Obtaining the iterator is also done in the synchronized block. All modifications must be made through the iterator. Iterator methods are used to process the synchronized list, including removing elements. Keep in mind that `next()` and `remove()` methods of the iterator work in lockstep. The current element from each iteration can be processed safely. This idiom also applies when using a `for(:)` loop for serial access, as this loop internally is translated to an iterator.

The program in **Example 23.16** is run with three threads that execute the `Runnable` `eliminator`. With manual synchronization on the synchronized list at (3), the threads execute as expected. One of the threads removed the year 2021 from the list and thus modified the list. As there was only one occurrence of the year 2021, the other two threads did not modify the list. With no manual synchronization on the synchronized list, the results are unpredictable. Most likely one or more exceptions will be thrown.

Regardless of manually synchronizing on the synchronized collection, as in **Example 23.16**, any modification made directly on the *underlying* collection during serial access will result in the runtime `java.util.ConcurrentModificationException`.

Example 23.16 *Serial Access in Synchronized Views of Collections*

Click here to view code image

```
package synced;
import java.util.*;
import java.util.stream.IntStream;

public class SerialAccessThreads  {

  public static void main(String[] args) throws InterruptedException {
    List<Integer> years = Collections.synchronizedList(new ArrayList<>()); // (1)
    years.add(2024); years.add(2023); years.add(2021); years.add(2022);

    Runnable eliminator = () -> {                                          // (2)
      boolean found = false;
      synchronized(years) {                                                // (3)
        Iterator<Integer> iteratorA = years.iterator();                    // (4)
        while (iteratorA.hasNext()) {
          if (iteratorA.next().equals(2021)) {                             // (5)

            iteratorA.remove();                                            // (6)
            found = true;
          }
        }
      }                                                                    // (7)
      System.out.println("List modified: " + found);
    };
    IntStream.rangeClosed(1, 3).forEach(i -> new Thread(eliminator).start());
  }
}
```

Probable output from the program with manual synchronization for serial access:

```
List modified: false
List modified: true
List modified: false
```

Probable output from the program without manual synchronization for serial access (comment out lines at (2) and (7)) (*output edited to fit on the page*):

Click here to view code image

```
Exception in thread "Thread-1"
Exception in thread "Thread-2"
Exception in thread "Thread-0" java.lang.NullPointerException:
    Cannot invoke "java.lang.Integer.equals(Object)" because the return value of
    "java.util.Iterator.next()" is null
    ...
```

Compound Mutually Exclusive Operations

A compound operation that requires multiple operations on a synchronized collection is not guaranteed to be thread-safe just because the individual operations are mutually exclusive. Concurrent threads executing such a compound operation can interleave the individual operations, as in the case of a compound arithmetic operator. **Example 23.17** illustrates implementing a compound mutually exclusive operation on a synchronized list using coarse-grained synchronization.

The class `DoubleAct` has a synchronized list (`names`) declared at (1), and provides two methods `add()` and `removeFirst()` at (2) and (3) to maintain this list.

The `Client` class uses the `DoubleAct` class. An instance of the `DoubleAct` class is populated at (10) in the `main()` method. A `Runnable` (`remover`) is implemented by the lambda expression at (11). It calls the `removeFirst()` method at (12) and prints the retrieved name. Three threads are created at (13) that execute the `Runnable remover`, calling the `removeFirst()` method on the shared `DoubleAct` instance. Since there are only two elements in the list, two of the threads should print one name each, and one of them should return `null`.

The method `removeFirst()` at (3) uses two mutually exclusive methods of the synchronized list. The intent is to return the element at index 0 if the list is not empty; otherwise, the value `null`. The method `size()` returns the current size of the synchronized list, and the method `remove()` deletes the element at index 0. The calls to these methods can be interleaved by concurrent threads executing the `removeFirst()` method between the time the list size is determined and the first element is removed, unless appropriate steps are taken.

Click here to view code image

```
// Thread t1                                  // Thread t2
...                                           ...
if (name.size() > 0) // true                  ...

...                                           if (name.size() > 0) // true

return remove(0);    // Size 0                ...
```

```
...                                          return remove(0);    // Exception!
```

The recommended solution is to use a synchronized block on the synchronized list, as shown at (4), to guarantee that the method `removeFirst()` has exclusive access to the synchronized list, until the individual exclusive operations have completed. **Example 23.17** shows output for both scenarios that we have sketched above. Note that the reentrant nature of the intrinsic lock on the synchronized list allows nested locking on the list for the individual exclusive operations.

Example 23.17 *Compound Mutually Exclusive Operation*

Click here to view code image

```
package synced;
import java.util.*;

public class DoubleAct {
  // Synchronized list:                                                    (1)
  private List<String> names = Collections.synchronizedList(new ArrayList<>());

  public void add(String name) { names.add(name); }                        // (2)

  public String removeFirst() {                                            // (3)
    synchronized(names) {                                                  // (4)
      if (names.size() > 0) {                                              // (5)
        try { Thread.sleep(1); }                                           // (6)
        catch(InterruptedException e) { e.printStackTrace(); }
        return names.remove(0);                                            // (7)
      } else { return null; }                                              // (8)
    }                                                                      // (9)
  }
}
```

Click here to view code image

```
package synced;
import java.util.stream.IntStream;

public class Client {                                                      // (9)
  public static void main(String[] args) {
    DoubleAct da = new DoubleAct();
    da.add("Laurel"); da.add("Hardy");                                     // (10)
    Runnable remover = () -> {                                             // (11)
      String name = da.removeFirst();                                      // (12)
      System.out.println(name);
    };
```

```
    IntStream.rangeClosed(1, 3).forEach(i -> new Thread(remover).start());// (13)
  }
}
```

Probable output from the program:

```
Laurel
Hardy
null
```

Probable output from the program when (4) and (9) are commented out (*output edited to fit on the page*):

Click here to view code image

```
Laurel
Hardy
Exception in thread "Thread-1" java.lang.IndexOutOfBoundsException:
      Index 0 out of bounds for length 0
      at ...
      at synced.DoubleAction.removeFirst(DoubleAction.java:15)
      at ...
```

23.7 Concurrent Collections and Maps

Most of the collections in the `java.util` package are not thread-safe—executing concurrent operations on them is courting disaster. Exceptions to this are the legacy `Vector` and `Hashtable` classes, but these use a single lock, allowing only one thread at a time to execute an operation on the collection while other threads are blocked. The *synchronized collections* created by methods in the `java.util.Collections` class are not much better, as they are just thread-safe wrappers around a backing collection, again providing only mutually exclusive operations on the backing collection. However, the *concurrent collections* provided by the `java.util.concurrent` package use special-purpose locking mechanisms to enable multiple threads to operate concurrently, without explicit locking on the collection and with minimal contention. The concurrent collections offer greater flexibility and higher scalability compared to the collections in the `java.util` package, when concurrent access is crucial for multiple threads sharing objects stored in a collection.

We assume familiarity with the Collections Framework, especially the core interfaces in the `java.util` package (**Chapter 15**, **p. 781**), as these are implemented by the concurrent collections to provide thread-safety and atomicity of operations defined by these interfaces:

`Collection`	(§15.3, p. 863)	`Queue`	(§15.6, p. 877)
`List`	(§15.3, p. 863)	`Deque`	(§15.7, p. 884)
`Set`	(§15.4, p. 867)	`Map`	(§15.8, p. 894)
`NavigableSet`	(§15.5, p. 874)	`NavigableMap`	(§15.10, p. 909)

It is also important to note that concurrent collections avoid memory consistency errors by defining a happens-before relationship that essentially states that an action by a thread to place an object into a concurrent collection happens-before a subsequent action to access or remove that object from the collection by another thread.

The concurrent collections and maps in the `java.util.concurrent` package can be categorized as follows:

- *Concurrent collections (p. 1485)*
- *Concurrent maps (p. 1490)*
- *Blocking queues (p. 1495)*
- *Copy-on-write collections (p. 1501)*

The following aspects about collections and maps should be noted, as they can be crucial in selecting an appropriate collection for maintaining the elements:

- *Non-blocking or blocking*
  Operations on a *non-blocking collection* do not use any locking mechanism (are *lock-free*) to access elements of the collection, allowing multiple threads to execute concurrently, in contrast to a *blocking collection* in which operations can block to acquire a lock before they can proceed, because only one thread at a time is allowed.
  Collections with `Concurrent` as a prefix to their name are all non-blocking—for example, a `ConcurrentHashMap`. All queues implementing the `BlockingQueue<E>` interface, not surprisingly, are blocking collections. The names of many queues reveal their blocking nature—for example, a `LinkedBlockingQueue`.
- *Unbounded and bounded*
  A *bounded collection* has a *fixed capacity* that defines the maximum number of elements allowed in the collection and is usually specified when the collection is created. When the collection is full, some operations, such as adding an element, will block until there is space in the collection. *Unbounded collections* are not bounded by any capacity restriction. *Optionally bounded collections* can be instantiated to behave either as bounded or as unbounded.
  An `ArrayBlockingQueue` is a bounded queue, whereas a `ConcurrentLinkedQueue` is unbounded, but a `LinkedBlockingDeque` can be optionally bounded.
- *The `null` value*

Some collections allow the `null` value, and others do not. All concurrent collections, blocking queues, and concurrent maps do *not* allow the `null` value as elements, whereas the copy-on-write collections do.

- *Duplicate elements*
  All collections that embody the concept of a set do not allow duplicates—for example, a `CopyOnWriteArraySet`. All maps (e.g., a `ConcurrentHashMap`) do not allow duplicate keys which would violate the concept of hashing values. However, all queues allow duplicates—for example, a `LinkedBlockingDeque`.
- *Ordering*
  Elements in some concurrent sets and maps do not have any ordering—for example, a `CopyOnWriteArraySet` or a `ConcurrentHashMap`.
  Some sets and maps keep their elements and entries in *sort order*, according to either their natural ordering or a total ordering defined by a comparator on the elements or the keys, respectively. Examples of concurrent sorted collections are a `ConcurrentSkipListSet` and a `ConcurrentSkipListMap`.
  Insertion ordering is the order in which the elements are inserted into the collection—as used by a `CopyOnWriteArrayList` for its elements.
  FIFO (*First-In-First-Out*) order is maintained by some thread-safe queues, such as a `LinkedBlockingQueue` and a `ConcurrentLinkedQueue`. Other queues have a more specialized ordering, like a `DelayQueue`.
  Not surprisingly, some thread-safe deques exhibit both FIFO and LIFO (*Last-In-First-Out*) order for their elements depending on how the deque is used—for example, a `LinkedBlockingDeque`.
- *Iterator behavior during serial access*
  Serial access is implemented differently for different categories of concurrent collections. It is important to understand the behavior of the iterator provided by these collections. An iterator has to address situations where modifications are made to the collection, outside the control of the iterator, that might render the state of the collection inconsistent—for example, removing or inserting elements during iteration. Iterators can be classified according to the response they provide in this situation, which is referred to as *concurrent modification*:
  - *Weakly consistent iterator*
    Most of the concurrent collections, and especially queues, provide an iterator that is *weakly consistent* during serial access. Such an iterator is created on a clone of the collection, and will always traverse elements that existed at the time the iterator was constructed only once. It *may not* reflect all subsequent modifications made to the collection after the iterator is constructed. It will also not throw a `ConcurrentModificationException`, and can execute concurrently with other operations on the collection. Weakly consistent iterators also provide guarantees against repeated elements occurring, and guard against a variety of errors and from infinite loops occurring during traversal.
    Weakly consistent iterators are also informally referred to as *fail-safe* iterators.
  - *Snapshot-style iterator*

A `CopyOnWriteArrayList`, which is based on a thread-safe version of an `ArrayList`, uses *snapshot-style iterators* for serial access. The name of the iterator reflects that it iterates over a snapshot of the underlying array taken at the time the iterator is created. Concurrent multiple *read* operations on this array snapshot are thread-safe as this snapshot of the array cannot be changed, but any *write* operation is done on a *new copy* of the array. An iterator is always created on the most current modified array. A snapshot iterator does *not* reflect any changes made in the copy of the array—in contrast to a weakly consistent iterator that may reflect such changes.

- *Fail-fast iterator*
  At the start of each iteration, a fail-fast iterator detects whether the collection has been modified. If that is the case, it throws a `ConcurrentModification-Exception`. In other words, it fails as soon as possible at the next iteration. It is perfectly safe during iteration to use the `Iterator.remove()` method to delete the current element, but not with the `Collection.remove()` method.
  Thread-unsafe collections from the `java.util` package exhibit this behavior, as do the `PriorityBlockingQueue` and `DelayDeque` classes in the `java.util.concurrent` package.

Concurrent Collections

Concurrent sets, queues, and deques are implemented by the classes `Concurrent-SkipListSet`, `ConcurrentLinkedQueue`, and `ConcurrentLinkedDeque`, respectively. These thread-safe classes implement the corresponding `NavigableSet`, `Queue`, and `Deque` interfaces in the `java.util` package, as shown in **Figure 23.4** and summarized in **Table 23.5**. Their characteristics are summarized in **Table 23.6**. These concurrent collections are *unbounded* and *non-blocking*. They do not allow `null` values and have a weakly consistent iterator. As they are unbounded, their insert operation will always succeed. Also worth noting is that their *bulk operations* (`addAll()`, `removeIf()`, `forEach()`) are not guaranteed to perform atomically.

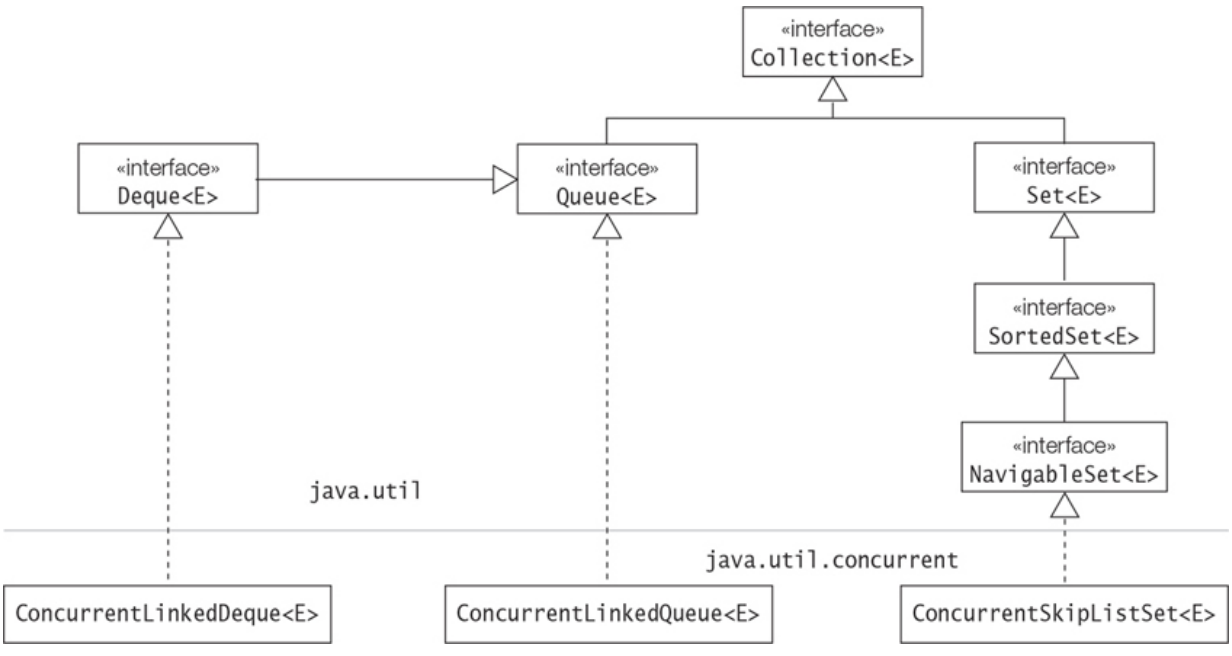

Figure 23.4 *Concurrent Collections in the* `java.util.concurrent` *Package*

Table 23.5 *Concurrent Collections in the* `java.util.concurrent` *Package*

Concurrent collections	Description
`ConcurrentSkipListSet<E> implements NavigableSet<E>`	An unbounded, non-blocking, concurrent `NavigableSet` implementation that internally uses a `ConcurrentSkipListMap`. Elements are sorted according to the natural ordering or by a comparator that is provided in the constructor.
`ConcurrentLinkedQueue<E> implements Queue<E>`	An unbounded, non-blocking, concurrent queue based on linked nodes.
`ConcurrentLinkedDeque<E> implements Deque<E>`	An unbounded, non-blocking, concurrent deque based on linked nodes.

Table 23.6 *Characteristics of Concurrent Collections*

Concurrent collections	`null` value	Duplicates	Ordering	Kind of iterator
`ConcurrentSkipListSet<E> implements NavigableSet<E>`	No	No	Sort order	Weakly consistent
`ConcurrentLinkedQueue<E> implements Queue<E>`	No	Yes	FIFO order	Weakly consistent
`ConcurrentLinkedDeque<E> implements Deque<E>`	No	Yes	LIFO/FIFO order	Weakly consistent

The `ConcurrentSkipListSet<E>` *Class*

The `ConcurrentSkipListSet` class implements the `NavigableSet` interface in the `java.util` package (§15.5, p. 811). The class implements a scalable, concurrent, sorted set. It provides efficient insertion, removal, and access operations that can be executed safely and concurrently by multiple threads. Its salient properties are summarized in

Table 23.6. Selected methods provided by the `ConcurrentSkipListSet` class are listed below.

```
// First-last elements
E pollFirst()
E pollLast()
```

Remove the first and the last elements currently in this concurrent set, respectively.

Click here to view code image

```
// Range-view operations
NavigableSet<E> headSet(E toElement,   boolean inclusive)
NavigableSet<E> tailSet(E fromElement, boolean inclusive)
NavigableSet<E> subSet(E fromElement, boolean fromInclusive,
                       E toElement,   boolean toInclusive)
```

Return different views of the underlying concurrent set, depending on the bound elements.

```

// Closest-matches
E ceiling(E e)
E floor(E e)
E higher(E e)
E lower(E e)
```

Determine closest-match elements according to various criteria.

Click here to view code image

```
// Reverse order
Iterator<E> descendingIterator()
NavigableSet<E> descendingSet()
```

The first method returns a reverse-order iterator for this concurrent set. The second method returns a reverse-order view of the elements in the set.

The `ConcurrentLinkedQueue<E>` *Class*

The `ConcurrentLinkedQueue` class implements the `Queue` interface, as shown in Figure 23.6. No new methods are defined, and existing methods of the `Queue` interface (§15.6, p.

814) are implemented as shown in Table 23.7. Note that since it is an unbounded queue, the insert operations will always succeed.

The `ConcurrentLinkedDeque<E>` *Class*

The `ConcurrentLinkedDeque` class implements the `Deque` interface, as shown in Figure 23.6. No new methods are defined, and existing methods of the `Deque` interface (§15.7, p. 821) are implemented as shown in Table 23.8. Note that since it is an unbounded deque, the insert operations will always succeed. Methods inherited from the `Queue` interface are marked with an asterisk (`*`) in Table 23.8.

Table 23.7 *Selected Methods in the* `ConcurrentLinkedQueue` *Class*

Operation	Throws exception	Returns special value
Insert at the tail	`add(e)` *will never throw* IllegalArgumentException	`offer(e)` *will never return* `false`
Remove from the head	`remove()` *can throw* `NoSuchElementException`	`poll()` *returns* `null` *if empty*
Examine element at the head	`element()` *can throw* `NoSuchElementException`	`peek()` *returns* `null` *if empty*

Table 23.8 *Selected Methods in the* `ConcurrentLinkedDeque` *Class*

Insert at the head	Insert at the tail	Runtime behavior on failure
`offerFirst(e)`	`offerLast(e)`, `offer(e)*`	*Never returns* `false`
`addFirst(e)`	`addLast(e)`, `add(e)*`	*Never throws* `IllegalStateException`
Remove from the head	**Remove from the tail**	**Runtime behavior on failure**
`pollFirst()`, `poll()*`	`pollLast()`	*Returns* `null` *if empty*

Insert at the head	Insert at the tail	Runtime behavior on failure
`removeFirst()`, `remove()*`	`removeLast()`	*Throws* `NoSuchElementException`
Examine at the head	**Examine at the tail**	**Runtime behavior on failure**
`peekFirst()`, `peek()*`	`peekLast()`	*Returns* `null` *if empty*
`getFirst()`, `element()*`	`getLast()`	*Throws* `NoSuchElementException`

Example 23.18 illustrates the iterator of a `ConcurrentSkipListSet` (declared at (1)) that is weakly consistent during traversal. The idea is to have two tasks operating on the concurrent sorted set: one repeatedly creating an iterator to traverse the elements of the collection and summing them (2), and another continuously removing elements from the set (5).

The utility class `ConcUtil` declares the auxiliary method `snooze()` that is used to put a thread to sleep. If interrupted, the exception is caught and the interrupt reinstated so that the caller of the `snooze()` method can take the appropriate action. This method is also used in other examples in this section.

The `Runnable sumValues` at (2) repeatedly creates an iterator at (3) to sum the values in the set at the time the iterator is created, and prints the result. It snoozes a little after each traversal of the set. The thread determines at (4) whether it has been interrupted. If interrupted, the infinite loop terminates, thereby terminating the thread.

The `Runnable removeValues` at (5) polls the set and prints the value, snoozing after each polling operation. It also detects whether it has been interrupted, terminating the infinite loop at (6) if that is the case.

The `main()` method instantiates and initializes a `ConcurrentSkipListSet` at (7) with random numbers between 0 and 1000. The two tasks are submitted to a service executor at (8). The main thread snoozes a little to allow the tasks to run. The call to the `shutdownNow()` method at (10) leads to any thread running to be interrupted, and the threads taking appropriate action to terminate their execution as described above.

The output from the program shows that only elements that are in the `ConcurrentSkipListSet` instance at the time the iterator is created contribute to the sum, and the sum drops correctly as elements are removed.

Example 23.18 *Concurrent Collections*

Click here to view code image

```
package concurrent;
import java.util.concurrent.TimeUnit;

public class ConcUtil {
  public static void snooze(int timeout, TimeUnit unit) {
    String threadName = Thread.currentThread().getName();
    try {
      unit.sleep(timeout);
    } catch (InterruptedException ex) {
      System.out.println(threadName + ": " + ex);
      Thread.currentThread().interrupt();         // Reinstate interrupt status.
    }
  }
}
```

Click here to view code image

```
package concurrent;
import java.util.*;
import java.util.concurrent.*;

public class ConcurrentSkipListSetDemo {

  private static ConcurrentSkipListSet<Integer> set;                    // (1)

  private static Runnable sumValues = () -> {                           // (2)
    String threadName = Thread.currentThread().getName();
    while (true) {
      int sum = 0;
      for (Integer v : set) {                                           // (3)
        sum += v;
      }
      System.out.printf(threadName + ": sum%9d%n", sum);
      ConcUtil.snooze(2, TimeUnit.SECONDS);
      if (Thread.interrupted()) break;                                  // (4)
    }
  };

  private static Runnable removeValues = () -> {                        // (5)
    String threadName = Thread.currentThread().getName();
    while (true) {
      Integer value = set.pollFirst();
      if (value == null) continue;
      System.out.printf(threadName + ": removed%5d%n", value);
      ConcUtil.snooze(2, TimeUnit.SECONDS);
```

```
        if (Thread.interrupted()) break;                                    // (6)
      }
    };

    public static void main(String[] args) {
      // Create and populate the set:                                       (7)
      set = new ConcurrentSkipListSet<>();
      new Random().ints(10, 0, 1000).forEach(val -> set.add(val));
      System.out.println(set);

      // Create an executor service to execute two tasks:                   (8)
      ExecutorService exs = Executors.newFixedThreadPool(2);
      try {
        exs.submit(sumValues);
        exs.submit(removeValues);

        ConcUtil.snooze(5, TimeUnit.SECONDS);                               // (9)
      } finally {
        System.out.println("Shutting down now.");
        exs.shutdownNow();                                                  // (10)
      }
    }
  }
```

Probable output from the program:

Click here to view code image

```
[20, 100, 236, 263, 299, 359, 548, 552, 591, 686]
pool-1-thread-1: sum     3654
pool-1-thread-2: removed   20
pool-1-thread-1: sum     3634
pool-1-thread-2: removed  100
pool-1-thread-1: sum     3534
pool-1-thread-2: removed  236
Shutting down now.
pool-1-thread-1: java.lang.InterruptedException: sleep interrupted
pool-1-thread-2: java.lang.InterruptedException: sleep interrupted
```

Concurrent Maps

Figure 23.5 shows the inheritance hierarchy of concurrent maps in the `java.util.concurrent` package. Note the interfaces `ConcurrentMap<K,V>` and `ConcurrentNavigableMap<K,V>` that extend the thread-unsafe map interfaces in the `java.util` package. Their implementations, `ConcurrentHashMap<K,V>` and `ConcurrentSkipListMap<K,V>`, provide efficient unsorted and sorted concurrent maps,

respectively. The concrete map implementations are summarized in **Table 23.9**, together with their characteristics in **Table 23.10**.

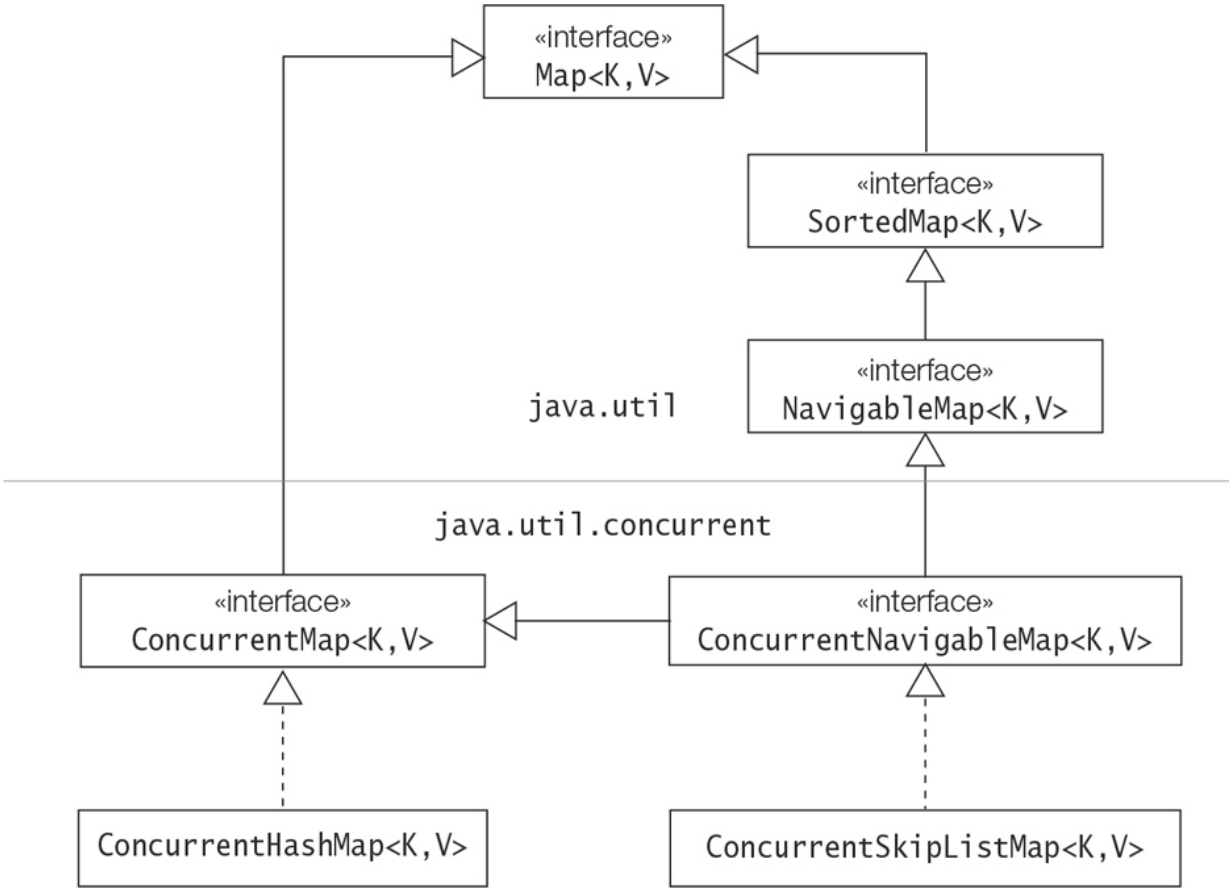

Figure 23.5 *Concurrent Maps in the* `java.util.concurrent` *Package*

Table 23.9 *Concurrent Maps in the* `java.util.concurrent` *Package*

Concurrent maps	Description
`interface ConcurrentMap<K,V> extends Map<K,V>` `ConcurrentHashMap<K,V> implements ConcurrentMap<K,V>`	A `Map` providing thread-safety and atomicity guarantees A hash table implementation supporting full concurrency of retrievals and high expected concurrency for updates
`interface ConcurrentNavigableMap<K,V> extends ConcurrentMap<K,V>, NavigableMap<K,V>` `ConcurrentSkipListMap<K,V> implements ConcurrentNavigableMap<K,V>`	A `ConcurrentMap` supporting `NavigableMap` operations, and recursively so for its navigable sub-maps A scalable concurrent `ConcurrentNavigableMap` implementation that is highly efficient for traversal

Table 23.10 *Characteristics of Concurrent Maps*

Concurrent maps	`null` value	Duplicates	Ordering	Kind of iterator
`ConcurrentHashMap<K,V> implements ConcurrentMap<K,V>`	Not allowed as key or values	Unique keys	No order	Weakly consistent
`ConcurrentSkipListMap<K,V> implements ConcurrentNavigableMap<K,V>`	Not allowed as key or values	Unique keys	Key sort order	Weakly consistent

The `ConcurrentMap<K,V>` *Interface*

In order to maintain the guarantees of thread-safety and atomicity of operations, the `ConcurrentMap<K,V>` overrides the methods shown below from the `Map<K,V>` interface, and stipulates that concurrent implementations will honor these guarantees—as exemplified by the `ConcurrentHashMap<K,V>` class, affording efficient concurrent insertion and lookup. Details on these overridden methods can be found in the `Map<K,V>` interface (§15.8, p. 831).

The implementation of the `ConcurrentHashMap<K,V>` class conceptually divides the map into *segments* (also called *sub-maps*) that can be independently locked by a thread, thus allowing several threads to perform operations on the map concurrently.

Click here to view code image

```
interface ConcurrentMap<K,V> extends Map<K, V>  // §15.8, p. 831.

default V compute(K key,
                  BiFunction<? super K,? super V,? extends V> remapFunction)
default V computeIfAbsent(K key,
                          Function<? super K,? extends V> mappingFunction)
default V computeIfPresent(K key,
                BiFunction<? super K,? super V,? extends V> remapFunction)

default void forEach(BiConsumer<? super K,? super V> action)
```

```
default V getOrDefault(Object key, V defaultValue)

default V merge(K key, V value,
                BiFunction<? super V,? super V,? extends V> remapFunction)
V putIfAbsent(K key, V value)
boolean remove(Object key, Object value)

V       replace(K key, V value)
boolean replace(K key, V oldValue, V newValue)
default void replaceAll(BiFunction<? super K,? super V,? extends V> func)
```

The `ConcurrentNavigableMap<K,V>` *Interface*

A concurrent, sorted map is defined by the `ConcurrentNavigableMap<K,V>` interface that extends both the `ConcurrentMap<K,V>` and the `NavigableMap<K,V>` interfaces. It overrides the methods shown below from its superinterfaces. Details on these overridden methods can be found in the `ConcurrentMap<K,V>` interface (**p. 1491**) and in the `NavigableMap<K,V>` interface (**§15.10**, **p. 845**).

The `ConcurrentSkipListMap<K,V>` class is an efficient implementation of the `Concurrent-NavigableMap<K,V>` class. This implementation is based on a concurrent variant of a *skip list* that is organized as a hierarchy of linked lists with subsequences of elements that are sorted, allowing efficient traversal, but at the extra cost of insertions in the skip list.

Click here to view code image

```
interface ConcurrentNavigableMap<K,V> extends ConcurrentMap<K,V>,
                                              NavigableMap<K,V>
NavigableSet<K> descendingKeySet()
ConcurrentNavigableMap<K,V> descendingMap()
```

Return a reverse-order `NavigableSet` view of the keys or a reverse-order `Concur-rent-NavigableMap` of the entries contained in this map, respectively.

Click here to view code image

```
ConcurrentNavigableMap<K,V> headMap(K toKey)
ConcurrentNavigableMap<K,V> headMap(K toKey, boolean inclusive)
```

Return a view of the portion of this map whose keys are strictly less than `toKey` or are less than or equal to `toKey`, if `inclusive` is `true` is specified.

```
NavigableSet<K> keySet()
NavigableSet<K> navigableKeySet()
```

Return a `NavigableSet` view of the keys contained in this map. These methods are equivalent.

Click here to view code image

```
ConcurrentNavigableMap<K,V> subMap(K fromKey, K toKey)
ConcurrentNavigableMap<K,V> subMap(K fromKey, boolean fromInclusive,
                                   K toKey, boolean toInclusive)
```

Return a view of the portion of this map whose keys range from `fromKey`, inclusive, to `toKey`, exclusive, or from `fromKey` to `toKey`, where inclusion of the interval keys is explicitly specified.

Click here to view code image

```
ConcurrentNavigableMap<K,V> tailMap(K fromKey)
ConcurrentNavigableMap<K,V> tailMap(K fromKey, boolean inclusive)
```

Return a view of the portion of this map whose keys are greater than or equal to `fromKey`, or are greater than or equal to `fromKey`, if `inclusive` is `true` is specified.

Example 23.19 illustrates using the `ConcurrentHashMap<K,V>` class. A frequency map is created for rolling a dice at (3) and (4). A reader, declared as `diceResultsReader` at (1), repeatedly creates a new key set of the map and prints its contents. A remover, declared as `diceResultRemover` at (2), continuously removes the entry for a random dice result from the map. Both the reader and the remover take a snooze after printing the contents and removing an entry, respectively. All threads terminate on being interrupted, as the `shutdownNow()` method will initiate cancellation of all tasks at (6).

One reader and two removers are submitted to the service executor at (5). The output shows that the map contents reflect correctly that entries are removed and printed concurrently.

Example 23.19 *Using a Concurrent Map*

Click here to view code image

```
package concurrent;
import java.util.*;
```

```
import java.util.concurrent.*;
import java.util.function.Function;
import java.util.stream.Collectors;

public class ConcurrentHashMapDemo {

  private static ConcurrentHashMap<Integer, Long> map;
  public static final int NUM_OF_THROWS = 1000;

  private static Runnable diceResultsReader = () -> {                    // (1)
    String threadName = Thread.currentThread().getName();
    while (true) {
      ConcurrentHashMap.KeySetView<Integer, Long> keySetView = map.keySet();
      String output = "";
      for (Integer key : keySetView) {
        Long value = map.get(key);
        output += " " + "<" + key + "," + value + ">";
      }
      System.out.println(threadName + ": {" + output + " }");
      ConcUtil.snooze(1000, TimeUnit.MILLISECONDS);
      if (Thread.interrupted()) break;
    }
  };

  private static Runnable diceResultRemover = () -> {                    // (2)
    String threadName = Thread.currentThread().getName();
    while (true) {
      ConcUtil.snooze(500, TimeUnit.MILLISECONDS);
      if (Thread.interrupted()) break;
      Integer key = ThreadLocalRandom.current().nextInt(1, 7);  // [1, 6]
      Long value = map.remove(key);
      if (value == null) continue;
      String removedEntry = threadName + ": removed "
                            + "<" + key + "," + value + ">";
      System.out.println(removedEntry);
    }
  };

  public static void main(String[] args) throws InterruptedException {
    map = new ConcurrentHashMap<>(6);                                    //  (3)
    new Random().ints(NUM_OF_THROWS, 1, 7)                               //  (4)
                .boxed()
                .parallel()
                .collect(Collectors.groupingByConcurrent(
                    Function.identity(),
                    () -> map,
                    Collectors.counting()));

    ExecutorService exs = Executors.newFixedThreadPool(3);
    try {
      exs.submit(diceResultsReader);                                     // (5)
```

```
        exs.submit(diceResultRemover);
        exs.submit(diceResultRemover);
        ConcUtil.snooze(5, TimeUnit.SECONDS);
      } finally {
        exs.shutdownNow();                                          // (6)
      }
    }
  }
```

Probable output from the program:

Click here to view code image

```
pool-1-thread-1: { <1,160> <2,159> <3,170> <4,178> <5,158> <6,175> }
pool-1-thread-2: removed <1,160>
pool-1-thread-3: removed <6,175>
pool-1-thread-2: removed <5,158>
pool-1-thread-1: { <2,159> <3,170> <4,178> }
pool-1-thread-2: removed <4,178>
pool-1-thread-1: { <2,159> <3,170> }
pool-1-thread-1: { <2,159> <3,170> }
pool-1-thread-2: removed <2,159>
pool-1-thread-1: { <3,170> }
pool-1-thread-2: removed <3,170>
pool-1-thread-1: java.lang.InterruptedException: sleep interrupted
pool-1-thread-2: java.lang.InterruptedException: sleep interrupted
pool-1-thread-3: java.lang.InterruptedException: sleep interrupted
```

Blocking Queues

Queues (and deques) are the indisputable choice when choosing a collection to manage shared data in producer-consumer problems. The interfaces for thread-unsafe queues (`Queue`, `Deque`) in the `java.util` package have been enhanced in the `java.util.concurrent` package to provide a wide variety of blocking queues that are thread-safe (**Figure 23.6**). These thread-safe queues are *blocking* because a thread *blocks* when trying to add an element and the queue is full, or when trying to remove an element and the queue is empty.

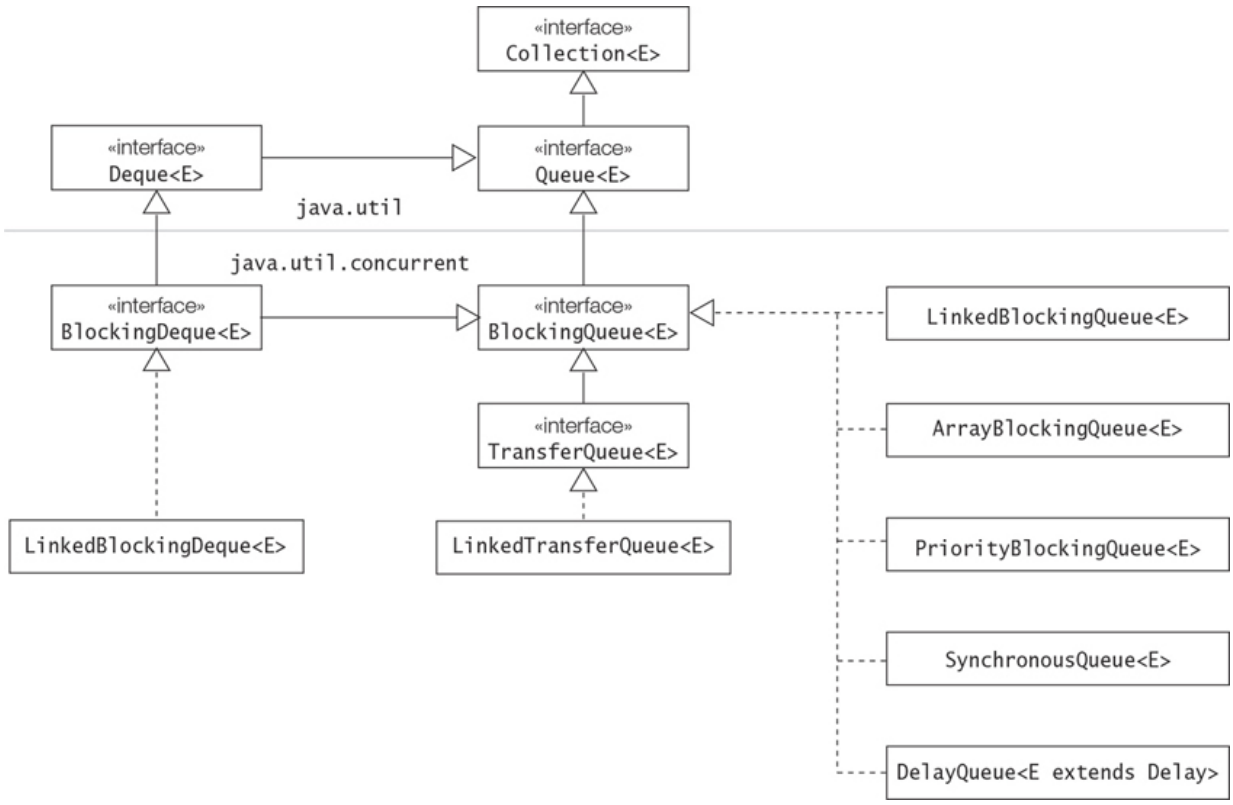

Figure 23.6 *Blocking Queues in the* `java.util.concurrent` *Package*

A summary of interfaces and classes that implement blocking queues and deques provided by the `java.util.concurrent` package is given in **Table 23.11**. Salient characteristics of these queues and deques are summarized in **Table 23.12**. None of the queues or deques allow a `null` value for an element, but they do allow duplicates. Being queues, FIFO ordering is the norm and a majority of them are weakly consistent when it comes to serial access.

Table 23.11 *Blocking Queues in the* `java.util.concurrent` *Package*

Blocking queues	Description
`interface BlockingQueue<E> extends Queue<E>` (**Table 23.13**)	A `Queue` that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.
`LinkedBlockingQueue<E> implements BlockingQueue<E>`	An optionally bounded blocking queue based on linked nodes.
`ArrayBlockingQueue<E> implements BlockingQueue<E>`	A bounded blocking queue backed by an array. A fairness policy can be specified so that threads blocked for insertion and removal are processed in FIFO order.

Blocking queues	Description
`PriorityBlockingQueue<E> implements BlockingQueue<E>`	An unbounded blocking queue that uses the same ordering rules as class `PriorityQueue` and supplies blocking retrieval operations. Implementation is based on a binary heap.
`SynchronousQueue<E> implements BlockingQueue<E>`	A blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa. No elements are stored.
`DelayQueue<E extends Delayed> implements BlockingQueue<E>`	An unbounded blocking queue of `Delayed` elements, in which an element can only be taken when its delay has expired. Elements implement the `java.util.concurrent.Delay` interface. Whether an element is available or not, it still counts toward the size of the queue. Implementation is based on a `PriorityQueue` instance that in turn uses a binary heap.
`interface TransferQueue<E> extends BlockingQueue<E>`	A `BlockingQueue` in which producers may wait for consumers to receive elements.
`LinkedTransferQueue<E> implements TranferQueue<E>`	An unbounded blocking `TransferQueue` based on linked nodes. The elements in the queue are ordered in FIFO order with respect to a given producer waiting for consumers.
`interface BlockingDeque<E> extends BlockingQueue<E>, Deque<E>` (**Table 23.14**)	A `Deque` that additionally supports blocking operations that wait for the deque to become nonempty when retrieving an element, and wait for space to become available in the deque when storing an element.
`LinkedBlockingDeque<E> implements BlockingDeque<E>`	An optionally bounded blocking deque based on doubly linked nodes.

Table 23.12 *Characteristics of Blocking Queues*

Blocking queues	`null` value	Duplicates	Ordering	Kind of iterator
`LinkedBlockingQueue<E>` implements `BlockingQueue<E>`	No	Allowed	FIFO order	Weakly consistent
`ArrayBlockingQueue<E>` implements `BlockingQueue<E>`	No	Allowed	FIFO order	Weakly consistent
`PriorityBlockingQueue<E>` implements `BlockingQueue<E>`	No	Allowed	Natural ordering	Fail-fast
`SynchronousQueue<E>` implements `BlockingQueue<E>`	No	Not applicable	Not applicable	Not applicable
`DelayQueue<E extends Delayed>` implements `BlockingQueue<E>`	No	Allowed	Longest-delayed order	Fail-fast
`LinkedTransferQueue<E>` implements `TranferQueue<E>`	No	Allowed	FIFO order	Weakly consistent
`LinkedBlockingDeque<E>` implements `BlockingDeque<E>`	No	Allowed	FIFO/LIFO order	Weakly consistent

The `BlockingQueue<E>` *Interface*

The `java.util.concurrent.BlockingQueue` interface extends the `java.util.Queue` interface, as shown in **Figure 23.6**. Compared to the operations shown for the `Queue` interface in **Table 23.7**, the `BlockingQueue` interface provides new insert and remove operations that can block or can time out—shown in the two rightmost columns in **Table 23.13**. Classes that implement the `BlockingQueue` interface are shown in **Table 23.11**, where general-purpose queues are listed at the top, down to more specialized thread-safe queues. The `LinkedBlockingQueue` class represents the all-round queue of choice in most cases.

Table 23.13 *Selected Methods in the* `BlockingQueue` *Interface*

Operation	Throws exception	Returns special value	Blocks	Times out
Insert at the tail	`add(e)` *can throw* `IllegalArgument-Exception`	`offer(e)` *returns* `true` *or* `false`	`put(e)` *blocks if no space*	`offer(e, time, unit)` *waits if necessary for specified time if no space*
Remove from the head	`remove()` *can throw* `NoSuchElement-Exception`	`poll()` *returns head element or* `null`	`take()` *blocks if empty*	`poll(time, unit)` *waits if necessary for specified time for head element, and returns* `null` *if no element becomes available*
Examine element at the head	`element()` *can throw* `NoSuchElement-Exception`	`peek()` *returns head element or* `null`	*Not applicable*	*Not applicable*

The `TransferQueue<E>` *Interface*

The `TransferQueue` interface extends the `BlockingQueue` interface to provide queues where producers wait for consumers to receive elements (**Figure 23.6**). The interface defines additional methods to utilize this property.

Click here to view code image

```
void transfer(E e)
boolean tryTransfer(E e)
boolean tryTransfer(E e, long timeout, TimeUnit unit)
```

The first method transfers the element to a consumer, waiting if necessary to do so.

The second method transfers the element to a waiting consumer immediately, if possible. It returns `true` if the element was transferred; otherwise, it returns `false`.

The third method transfers the element to a consumer if it is possible to do so before the timeout elapses. It returns `true` if successful, and `false` if the specified waiting time elapses before completion, in which case the element is not left enqueued.

Click here to view code image

```
boolean hasWaitingConsumer()
int getWaitingConsumerCount()
```

The first method returns `true` if there is at least one consumer waiting to receive an element via `BlockingQueue.take()` or a timed poll.

The second method returns an estimate of the number of consumers waiting to receive elements via `BlockingQueue.take()` or a timed poll.

The `BlockingDeque<E>` *Interface*

The `java.util.concurrent.BlockingDeque` interface extends both the `java.util.Deque` interface and the `BlockingQueue` interface, as shown in **Figure 23.6**. Compared to the operations shown for the `ConcurrentLinkedDeque` interface in **Table 23.8**, the `Blocking-Deque` interface provides new insert and remove operations, both at the head and tail of a deque, that can block or can time out—shown in the light-gray rows in **Table 23.14**. Methods inherited from the `BlockingQueue` interface are marked with an asterisk (`*`) in **Table 23.14**. The `LinkedBlockingDeque` class that implements the `BlockingDeque` interface is shown in **Table 23.11**. This class represents an all-round implementation of a blocking deque.

Table 23.14 *Selected Methods in the* `BlockingDeque` *Interface*

Insert at the head	Insert at the tail	Runtime behavior on failure
`offerFirst(e)`	`offerLast(e), offer(e)*`	*Returns* `false` *if full*
`addFirst(e)`	`addLast(e), add(e)*`	*Throws* `IllegalStateException`
`putFirst(e)`	`putLast(e), put(e)*`	*Blocks if full*
`offerFirst(e, time, unit)`	`offerLast(e,time,unit), offer(e, time, unit)*`	*Times out*

Insert at the head	Insert at the tail	Runtime behavior on failure
Remove from the head	Remove from the tail	Runtime behavior on failure
`pollFirst()`, `poll()*`	`pollLast()`	*Returns* `null` *if empty*
`removeFirst()`, `remove()*`	`removeLast()`	*Throws* `NoSuchElementException`
`takeFirst()`, `take()*`	`takeLast()`	*Blocks if empty*
`pollFirst(time, unit)`, `poll(time, unit)*`	`pollLast(time, unit)`	*Times out*
Examine at the head	**Examine at the tail**	**Runtime behavior on failure**
`peekFirst()`, `peek()*`	`peekLast()`	*Returns* `null` *if empty*
`getFirst()`, `element()*`	`getLast()`	*Throws* `NoSuchElementException`

Example 23.20 demonstrates using the `LinkedBlockingQueue` class that implements the `BlockingQueue` interface. A producer is defined at (2) that puts a fixed number of random values (between 0 and 100) in the blocking queue. Note that the `put()` call at (3) can block. The thread sleeps for a little while after each `put()` operation. When done, it puts a *poison value* (which is not a legal value) in the queue at (4), that is interpreted as no more values are in the queue.

A consumer is defined at (5) that continuously takes a value from the blocking queue at (7) and prints it. It sleeps for a little while after each `take()` operation that can block. Before taking a value, it checks at (6) to see if the value at the head of the queue is the poison value. If so, the infinite loop is terminated, and thereby the consumer as well.

The `LinkedBlockingQueue` class is instantiated at (8). Note that the blocking queue is bounded, but the number of values to put in the queue is greater than its capacity—which

can result in the `put()` operation to block if the queue is full. A service executor is created at (9). One producer and two consumers are submitted to the service executor at (10). The call to the `shutdown()` method at (11) allows threads to complete execution.

The output from the program shows that the producer puts the values in the blocking queue and the two consumers take the values and process them.

Example 23.20 *Linked Blocking Queue*

Click here to view code image

```
package concurrent;
import java.util.concurrent.*;

public class LinkedBlockingQueueDemo {

  public static final int UPPER_BOUND = 3;
  public static final int NUM_OF_VALUES = 5;
  public static final int STOP_VALUE = -1;

  private static BlockingQueue<Integer> queue;                          // (1)

  private static Runnable producer = () -> {                            // (2)
    String threadName = Thread.currentThread().getName();
    ThreadLocalRandom tlrng = ThreadLocalRandom.current();
    try {
      for (int i = 0; i < NUM_OF_VALUES; i++) {
        Integer value = tlrng.nextInt(100);
        queue.put(value);                                               // (3)
        System.out.println(threadName + ": put " + value);
        Thread.sleep(tlrng.nextInt(200));
      }
      queue.put(STOP_VALUE);                                            // (4)
      System.out.println(threadName + ": done.");
    } catch (InterruptedException ie) {
      ie.printStackTrace();
    }
  };

  private static Runnable consumer = () -> {                            // (5)
    String threadName = Thread.currentThread().getName();
    ThreadLocalRandom tlrng = ThreadLocalRandom.current();
    while (true) {
      try {
        Integer head = queue.peek();                                    // (6)
        if (head != null && head.equals(STOP_VALUE)) {
            System.out.println(threadName + ": done.");
            break;
        }
```

```
          Integer value = queue.take();                                    // (7)
          System.out.println(threadName + ": processing " + value);
          Thread.sleep(tlrng.nextInt(1000));
        } catch (InterruptedException ie) {
          ie.printStackTrace();
        }
      }
    };

    public static void main(String[] args) {
      queue = new LinkedBlockingQueue<>(UPPER_BOUND);                      // (8)
      ExecutorService exs = Executors.newFixedThreadPool(3);               // (9)
      try {
        exs.submit(producer);                                              // (10)
        exs.submit(consumer);
        exs.submit(consumer);
      } finally {
        exs.shutdown();                                                    // (11)
      }
    }
}
```

Probable output from the program:

Click here to view code image

```
pool-1-thread-1: put 43
pool-1-thread-2: processing 43
pool-1-thread-1: put 84
pool-1-thread-3: processing 84
pool-1-thread-1: put 60
pool-1-thread-3: processing 60
pool-1-thread-1: put 55
pool-1-thread-1: put 17
pool-1-thread-3: processing 55
pool-1-thread-1: done.
pool-1-thread-3: processing 17
pool-1-thread-2: done.
pool-1-thread-3: done.
```

Copy-on-Write Collections

The copy-on-write collections comprise special-purpose concurrent lists and sets that are recommended when read operations vastly outnumber mutative operations on the collection. The classes `CopyOnWriteArrayList` and `CopyOnWriteArraySet` implement the `List` and the `Set` interfaces in the `java.util` package (**Figure 23.7**). A summary of the copy-

on-write classes in the `java.util.concurrent` package is given in **Table 23.15**. Salient characteristics of these collections are summarized in **Table 23.16**.

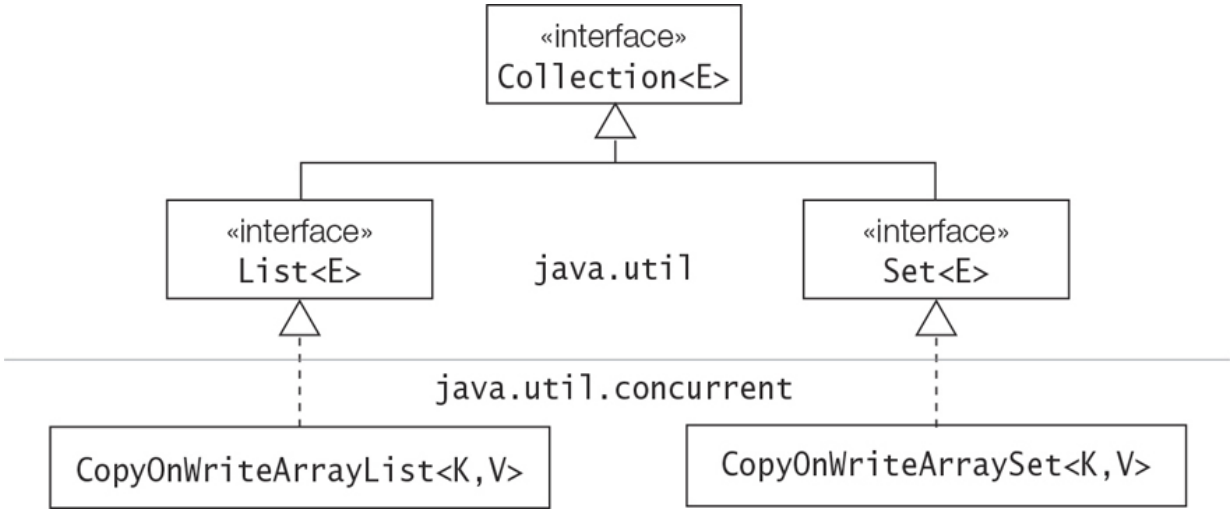

Figure 23.7 *Copy-on-Write Collections in the* `java.util.concurrent` *Package*

Table 23.15 *Copy-on-Write Collections in the* `java.util.concurrent` *Package*

Copy-on-write collections	Description
`CopyOnWriteArrayList<E> implements List<E>`	A thread-safe variant of `ArrayList` in which all mutative operations (add, set, etc.) are implemented by making a fresh copy of the underlying array.
`CopyOnWriteArraySet<E> implements Set<E>`	A `Set` that uses an internal `CopyOnWriteArrayList` for all of its operations.

Table 23.16 *Characteristics of Copy-on-Write Collections*

Copy-on-write collections	`null` value	Duplicates	Ordering	Kind of iterator
`CopyOnWriteArrayList<E> implements List<E>`	Yes	Yes	Insertion order	Snapshot-style
`CopyOnWriteArraySet<E> implements Set<E>`	Yes	No	No order	Snapshot-style

The `CopyOnWriteArrayList<E>` *Class*

The `CopyOnWriteArrayList` class implements the `java.util.List` interface (§15.3, p. 801), providing a thread-safe list that is recommended for a vast number of concurrent read and traversal operations. It uses an underlying array to implement its operations. By default, such a list is immutable, guaranteeing thread-safety of concurrent operations. Any mutative operation (add, set, remove, etc.) is done on a new *copy* of its entire underlying array which then supersedes the previous version for subsequent operations. For efficiency reasons, the list size should be kept small and modifications should be minimized, as they are expensive, incurring the copying cost of its underlying array and extra space.

Traversal via snapshot-style iterators is efficient and thread safe—there is no need for any external synchronization, since it relies on the immutable state of the underlying array at the time the iterator is constructed (**Example 23.21**). Note that immutability of the list during traversal rules out the `remove()` operation of the iterator, as this can invalidate its snapshot traversal guarantee if allowed. In the code below, the `Iterator.remove()` method at (2a) results in an `UnsupportedOperation-Exception`, whereas the `List.remove()` method is allowed since it will be performed on a new copy of the underlying array. Note that a thread-unsafe `ArrayList` instance would allow both (2a) and (2b).

Click here to view code image

```
List<Integer> cowlist = new CopyOnWriteArrayList<Integer>();          // (1a)
cowlist.addAll(Arrays.asList(10, 20, 30));
Iterator<Integer> iter = cowlist.iterator();
while (iter.hasNext()) {
  Integer i = iter.next();
  if (i == 20) {
//  iter.remove();                    // (2a) UnsupportedOperationException
    cowlist.remove(i);                // (2b) OK

    continue;
  }
  System.out.print(i + " ");          // With (2b): 10 30
}
```

In addition to the methods of the `List` interface, the following useful methods are also defined by the `CopyOnWriteArrayList` class:

```
boolean addIfAbsent(E e)
```

Appends the element, if not present.

Click here to view code image

```
int addAllAbsent(Collection<? extends E> c)
```

Appends to the end of this list all of the elements in the specified collection that are not already contained in this list.

The `CopyOnWriteArraySet<E>` *Class*

The `CopyOnWriteArraySet` class implements the `java.util.Set` interface (§15.4, p. 804). It does not implement any additional methods. Internally it uses a `CopyOnWriteArrayList`, and therefore shares the same basic properties with the list, except that, being a set, it does not allow duplicates and its elements have no ordering.

Example 23.21 illustrates the snapshot-style iterator of a `CopyOnWriteArrayList`. Such a list is created at (1) and populated with three values. A task to traverse the list and print its elements is defined by the `Runnable iter` at (2). Two threads are created at different times to execute this task at (4) and (6), respectively. The main thread modifies the list by adding a new value and removing a value after the start of the first thread. The output shows that the result from the first thread is not affected by the modifications done by the main thread, as its iterator only traverses those elements that were in the list when the iterator was created. The result from the second thread shows the state of the list when it was created—that is, after the list was modified by the main thread.

Example 23.21 *Copy-on-Write Array List*

Click here to view code image

```
package concurrent;
import java.util.*;
import java.util.concurrent.*;

public class CopyOnWriteArrayListDemo {

  public static void main(String[] args) {
    List<Integer> cowlist = new CopyOnWriteArrayList<Integer>();          // (1)
    cowlist.addAll(Arrays.asList(1, 2, 3));

    Runnable iter = () -> {                                               // (2)
      String threadName = Thread.currentThread().getName();
      for (Integer i : cowlist) {
        System.out.println(threadName + ": " + i);
        ConcUtil.snooze(1, TimeUnit.SECONDS);
      }
    };

    // First iterator:
```

```
        new Thread(iter, "Iterator A").start();                          // (4)

        // Snooze, add, and remove in main thread.                       // (5)
        ConcUtil.snooze(1, TimeUnit.SECONDS);
        Integer newValue = 4;
        cowlist.add(newValue);
        System.out.println("New value added: " + newValue);
        Integer first = cowlist.remove(0);
        System.out.println("Value removed:   " + first);

        // Second iterator:
        new Thread(iter, "Iterator B").start();                          // (6)
      }
    }
```

Probable output from the program:

```
Iterator A: 1
New value added: 4
Iterator A: 2
Value removed:   1
Iterator B: 2
Iterator A: 3
Iterator B: 3
Iterator B: 4
```

Review Questions

23.1 Which of the following statements is true?

Select the one correct answer.

a. A single thread executor service cannot be used to schedule tasks.

b. A fixed thread pool is configured with a given level of parallelism.

c. A scheduled executor service allows a task to be run with a specified delay.

d. A cached thread pool implements a work-stealing mechanism.

23.2 Which of the following statements is true?

Select the one correct answer.

a. The executor service's `shutdown()` method cancels all tasks that are still running.

b. The executor service's `shutdown()` method disables new tasks from being submitted.

c. The executor service's `awaitTermination()` method cannot be interrupted.

d. The executor service's `shutdownNow()` method awaits termination of tasks that are still running.

23.3 Which of the following statements is true about the `ReentrantReadWriteLock` class?

Select the one correct answer.

a. Its read lock prevents concurrent read operations from reading inconsistent data.

b. Its write lock prevents concurrent read operations from reading inconsistent data.

c. Its write lock acquisition cannot be interrupted.

d. Its read lock acquisition cannot be interrupted.

23.4 Given the following code:

Click here to view code image

```
var values = List.of("1", "2", "3", "4", "5", "6", "7", "8");
var list1 = new CopyOnWriteArrayList<Integer>();
for (String value: values) {
    new Thread(() -> {
        list1.add(Integer.valueOf(value));
    }).start();
}
System.out.println(list1);

var list2 = values.parallelStream().map(v->Integer.valueOf(v))
                                   .toList();
System.out.println(list2);

var list3 = new ArrayList<Integer>();
values.parallelStream().map(v->Integer.valueOf(v)).forEach(v->list3.add(v));
System.out.println(list3);
```

Which lists will always print the same values as in the initial list?

Select the one correct answer.

a. `list1`

b. `list2`

c. `list3`

d. `list1` and `list2`

e. `list2` and `list3`

f. `list1` and `list3`

g. All of the lists

h. None of the lists

23.5 Which of the following statements is true?

Select the one correct answer.

a. Synchronized collections provide a synchronized `iterator()` method.

b. Copy-on-write collections provide a synchronized `iterator()` method.

c. Immutable collections provide a synchronized `iterator()` method.

d. Synchronized collections require the programmer to implement a synchronized `iterator()` method.

e. Copy-on-write collections require the programmer to implement a synchronized `iterator()` method.

f. Immutable collections require the programmer to implement a synchronized `iterator()` method.

23.6 Which of the following statements is true about atomic variables?

Select the one correct answer.

a. Atomic variables represent the smallest indivisible unit of data.

b. Atomic variables ensure memory consistency using synchronized operations.

c. Atomic variables provide methods that can throw an `InterruptedException`.

d. Atomic variables do not utilize an intrinsic locking mechanism.

23.7 Given the following code:

Click here to view code image

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
public class RQ7 {
  public static void main(String[] args) {
    AtomicLong aValue = new AtomicLong(0);
    Callable<Number> c = () -> {
      AtomicLong value = aValue;
        return value.incrementAndGet();
      };

    Collection<Callable<Number>> cc = List.of(c, c, c);
    ExecutorService es = Executors.newFixedThreadPool(2);
    List<Future<Number>> futures = null;
    try {
      futures = es.invokeAll(cc);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    es.shutdown();

    futures.stream().mapToLong(v-> {
    try {
      return v.get().longValue();
    }catch(Exception e) {
      return -1;
    }}).forEach(v->System.out.print(v));
    System.out.print(aValue);
  }
}
```

Which of the following output can be produced by this program? Select the two correct answers.

a. `1233`

b. `2133`

c. `1122`

d. `2122`

e. `2123`

23.8 Given the following code:

Click here to view code image

```
import java.util.concurrent.locks.*;
public class RQ8 {
  public static void main(String[] args) {

    ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    Lock rl = rwl.readLock();
    Lock wl = rwl.writeLock();
    try {
      rl.lock();
      System.out.println("Read lock acquired");
      if (wl.tryLock()) {
        System.out.println("Write lock acquired");
      }
    } catch(Exception e) {
      System.out.println("Lock acquisition failed");
    } finally {
      rl.unlock();
      if (rwl.isWriteLocked()) {
        wl.unlock();
        System.out.println("Write lock released");
      }
    }
    System.out.println("The end");
  }
}
```

What is the result?

Select the one correct answer.

a. `Read lock acquired The end`

b. `Read lock acquired Write lock acquired Write lock released The end`

c. `Read lock acquired Lock acquisition failed The end`

d. `Lock acquisition failed The end`

e. `Lock acquisition failed Write lock released The end`

23.9 Given the following code:

Click here to view code image

```
public class RQ9 {
  private static volatile int counter = 10;
  public static void main(String[] args) {
    Runnable r = () -> counter--;
```

```
    while (counter > 0) {
      new Thread(r).start();
      System.out.print(counter);
    }
  }
}
```

What is the result?

Select the one correct answer.

a. Either `9876543210` or `876543210-1` or 987654321

b. Either `9876543210` or `10987654321` or 987654321

c. `9876543210`

d. The result is unpredictable.

23.10 Given the following code:

Click here to view code image

```
import java.util.concurrent.*;
public class RQ10 {
  public static void main(String[] args) {
    try {
      ExecutorService es = Executors.newFixedThreadPool(2);
      Future<?> f1 = es.submit(() -> "acme");
      Future<?> f2 = es.submit(() -> {}); // (1)
      es.shutdown();                      // (2)
      Object o1 = f1.get();
      Object o2 = f2.get();               // (3)
      System.out.println(o1 + " " + o2);  // (4)
    } catch (InterruptedException | ExecutionException e) {
      e.printStackTrace();
    }
  }
}
```

What is the result?

Select the one correct answer.

a. An exception is thrown at (1).

b. An exception is thrown at (2).

c. An exception is thrown at (3).

d. An exception is thrown at (4).

e. `acme` null

f. `acme`

23.11 Given the following code:

Click here to view code image

```
import java.util.*
import java.util.concurrent.*;
import java.util.stream.*;

public class RQ11 {
  public static void main(String[] args) {
    Map<Integer, String> map =
        new ConcurrentHashMap<>(Map.of(1,"a",2,"b",3,"c",4,"d",5,"e"));
    List<Future<String>> results = new CopyOnWriteArrayList<>();

    ExecutorService es = Executors.newFixedThreadPool(3);
    for (int i = 1; i <= map.size(); i++) {
      final int key = i;
      Future<String> f = es.submit(() -> map.get(key).toUpperCase());
      if (i % 2 != 0) {
        f.cancel(true);
      }

      results.add(f);
    }
    es.shutdown();
    String result = (results.stream().allMatch(r -> r.isDone()))
        ? results.stream().filter(r -> !r.isCancelled()).map(r -> {
            try {
              return r.get();
            } catch (InterruptedException | ExecutionException e) {
              return "X";
            }
          }).collect(Collectors.joining())
        : "Z";
    System.out.println(result);
  }
}
```

Which of the following statements are true about the result of this program? Select the two correct answers.

a. The program terminates with an exception.

b. The program only prints the letters `B` or `D`.

c. The program never prints the letters `B` or `D`.

d. The program only prints the letters `A`, `C`, or `E`.

e. The program never prints the letters `A`, `C`, or `E`.

f. The program may print the letters `A`, `B`, `C`, `D`, or `E`.

g. The program may print the letter `X`.

h. The program may print the letter `Z`.