Capitolul 13 - Concurrency
Introducing Threads

Thread

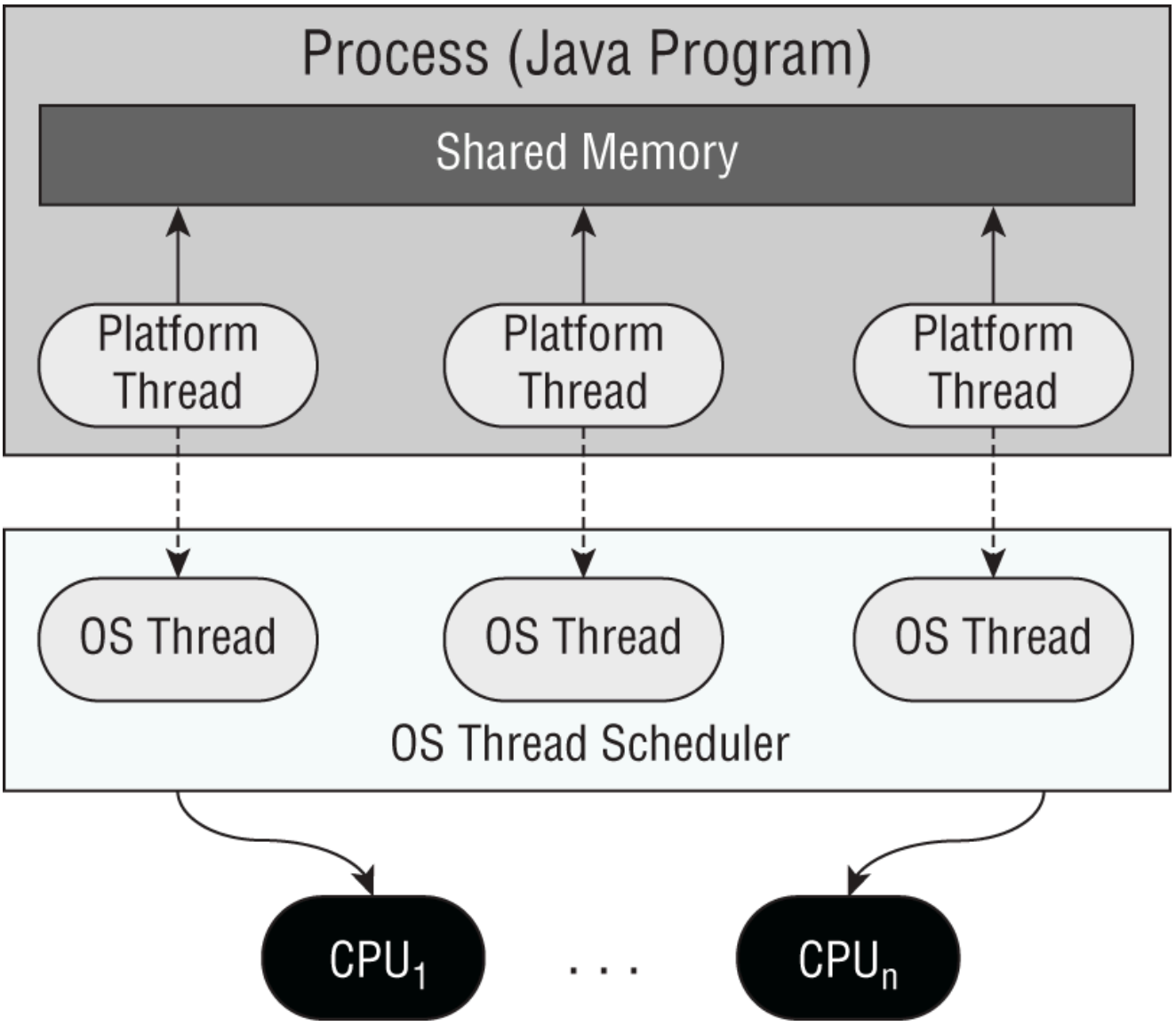
The smallest unit of execution that can be scheduled by the OS.

Process

A group of associated threads that execute in the same environment.

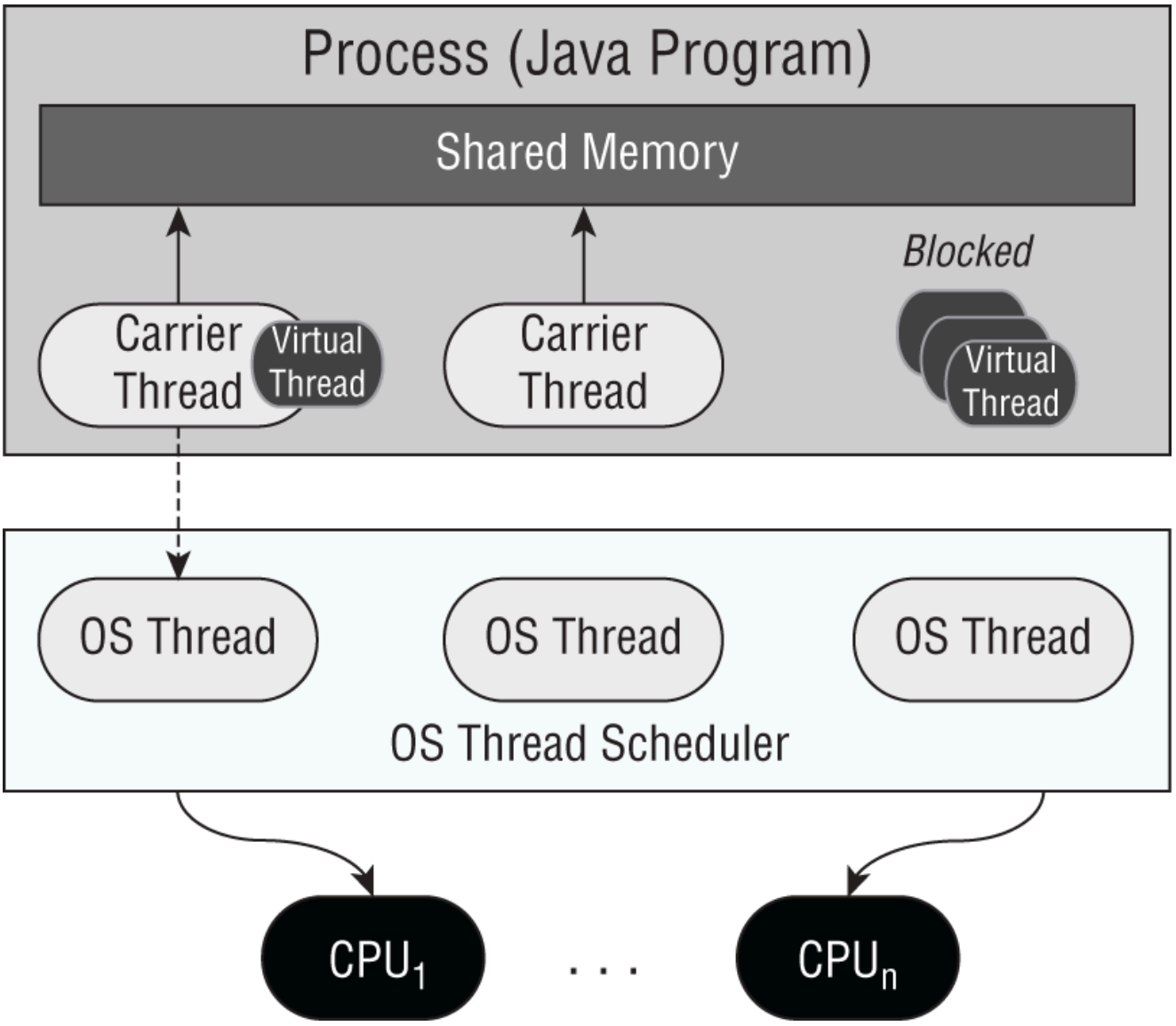
Shared Environment

The threads in the same process share the same memory space and can communicate directly with each other.



Shared memory

`Static` Variables as well as instance and local variables passed to a thread.



Understanding Thread Concurrency

Concurrency

Property of executing multiple threads and processes at the same time.

Thread scheduler

Determines which threads should be currently executing.

Context switch

The process of storing a thread's current state and later restoring the state of the thread to continue execution.

Thread priority

Is a numeric value associated with a thread that the thread scheduler considers when determining which threads should execute. The priority can be set from 1 (`Thread.MIN_PRIORITY`) to 10 (`Thread.MAX_PRIORITY`), either before a thread is started or while it is running.

```
var thread1 = new Thread(() -> System.out.print("Super
Important"));
thread1.setPriority(Thread.MAX_PRIORITY);
thread1.start();

var thread2 = new Thread(() -> System.out.print("Less
Important"));
thread2.start();
thread2.setPriority(2);
```

Creating a Thread

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

/*----*/

Thread.ofPlatform().start(() -> System.out.print("Hello"));
System.out.print("World");
```

TABLE 13.1 Creating and starting a `Thread`

Code	Type	Description
`var builder = Thread.ofPlatform();` `Thread thread = builder.start(runnable);`	Platform	Factory
`var builder = Thread.ofVirtual();` `Thread thread = builder.start(runnable);`	Virtual	Factory
`Thread thread = new Thread(runnable);` `thread.start();`	Platform	Constructor

Starting with Java 21, the factory method is preferable since it is clearer which type of thread you are getting. The factory method creates a builder, which allows you to call other methods to set attributes like the name.

For platform threads, you can set a priority via the builder object by calling `priority()`. The priority for virtual threads is always 5 (`Thread.NORM_PRIORITY`) and cannot be changed. Calling `setPriority()` on the newly created virtual `Thread` has no effect.

```
12: public static void main(String[] args)
13:       throws InterruptedException {
14:    Runnable printInventory =
15:       () -> System.out.println("Printing zoo inventory");
16:    Runnable printRecords = () -> {
17:       for (int i = 0; i < 3; i++)
18:          System.out.println("Printing record: " + i);
19:    };
20:    System.out.println("begin");
21:    var platformThread = Thread.ofPlatform()
22:       .priority(10)
23:       .start(printInventory);
24:    var virtualThread = Thread.ofVirtual()
25:       .start(printRecords);
26:    var constructorThread = new Thread(printInventory);
27:    constructorThread.start();
```

```
28:     System.out.println("end");
29:     platformThread.join();
30:     virtualThread.join();
31:     constructorThread.join();
32: }
```

The answer is that the order is unknown until runtime.
The `join()` methods on lines 29–31 tell the `main()` method not to end before the three threads have completed. The `join()` method throws an `InterruptedException` if it fails, which the `main()` method declares.
Be mindful of code that attempts to start a thread by calling `run()` instead of `start()`. Calling `run()` on a `Thread` or a `Runnable` *does not start a new thread*. While the following code snippet will compile, it runs synchronously rather than starting a thread:

```
new Thread(printInventory).run();
```

Working with Daemon Threads

Daemon thread

Is one that will not prevent the JVM from exiting when the program finishes.

A Java application terminates when the only threads that are running are daemon threads.

```
1:  public class Zoo {
2:     public static void pause() {                 // Defines
the thread task
3:        try {
4:           Thread.sleep(10_000);                  // Wait for
10 seconds
5:        } catch (InterruptedException e) {}
6:        System.out.println(“Thread finished!”);
7:     }
8:
9:     public static void main(String[] unused) {
10:       var job = Thread.ofPlatform().start(Zoo::pause);
11:       System.out.println(“Main method finished!”);
12:    } }
```

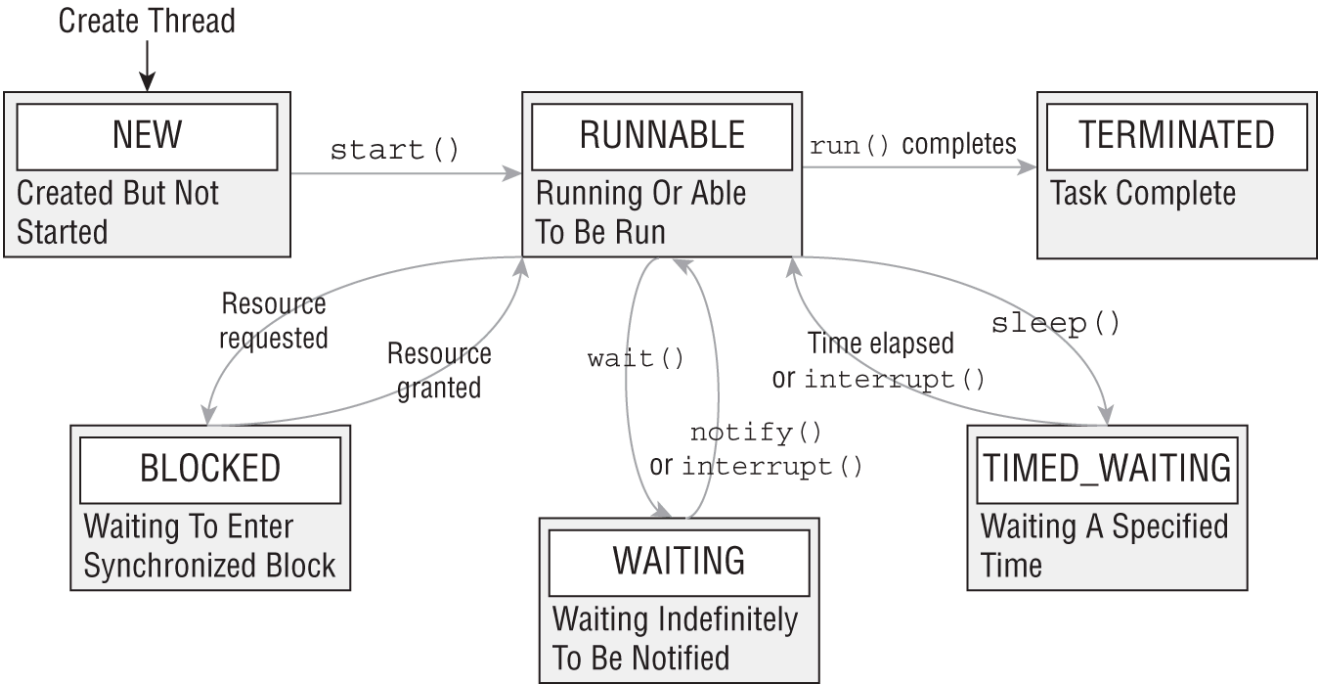
```
----

Main method finished!
< 10 second wait >
Thread finished!
```

That's right. Even though the `main()` method is done, the JVM will wait for the user thread to be done before ending the program.

```
10: var job = Thread.ofPlatform().daemon(true).start(Zoo::pause);
--------------
Main method finished!
```

Managing a Thread's Life Cycle




```
var thread = Thread.ofPlatform().start(() → {
   try {
      Thread.sleep(1000);
   } catch (InterruptedException e) {
      System.out.println("Interrupted!");
   }});
thread.interrupt();

//This code prints `Interrupted!` and then resumes running.
```

What if `interrupt()` is called on a thread that is already in the `RUNNABLE` state? In this case, an exception won't be thrown.

If an operation that is not supported is called on a thread, such as interrupting a suspended thread, an `IllegalThreadStateException` will be thrown.

TABLE 13.2 Java thread terminology

Term	Description
Carrier thread	System thread that runs virtual threads when they are not blocked
Daemon thread	Thread that will not prevent the JVM from exiting when the program finishes
Platform thread	Thread that is scheduled by the operating system
Process	Group of associated threads that execute in the same shared environment
System thread	Thread created by the JVM that runs in the background of the application, such as the garbage collector
Task	Single unit of work performed by a thread
Thread	Smallest unit of execution that can be scheduled
User-defined thread	Thread created by the application developer to accomplish a specific task
Virtual thread	Lightweight thread that is mapped to a carrier thread when needed to run

Creating Threads with the Concurrency API

The Concurrency API includes the `ExecutorService` interface, which defines services that create and manage threads.

Introducing the Single-Thread Executor

The Concurrency API includes the `Executors` factory class that can be used to create instances of the `ExecutorService` object.

```
try (ExecutorService service =
Executors.newSingleThreadExecutor()) {
   System.out.println("begin");
   service.execute(printInventory);
   service.execute(printRecords);
   service.execute(printInventory);
   System.out.println("end");
}
```

In this example, we use the `newSingleThreadExecutor()` method to create the service. Unlike our earlier example, in which we had four threads (one `main()` and three new threads), we have only two threads (one `main()`

and one new thread). This means that the output, while still unpredictable, will have less variation than before.
Notice that the `printRecords` loop is no longer interrupted by other `Runnable` tasks sent to the thread executor. With a single-thread executor, tasks are guaranteed to be executed sequentially. Notice that the `end` text is output while our thread executor tasks are still running. This is because the `main()` method is still an independent thread from the `ExecutorService`.

Submitting tasks

1. `execute()` method takes a `Runnable` instance and completes the task asynchronously. Return type: `void`;
2. `submit()` method, Return type: `Future`

TABLE 13.3 `ExecutorService` methods

Method name	Description
`void execute(Runnable command)`	Executes `Runnable` task at some point in future.
`Future<?> submit(Runnable task)`	Executes `Runnable` task at some point in future and returns `Future` representing task.
`<T> Future<T> submit(Callable<T> task)`	Executes `Callable` task at some point in future and returns `Future` representing pending results of task.
`<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)`	Executes given tasks and waits for all tasks to complete. Returns `List` of `Future` instances in the same order in which they were in original collection.
`<T> T invokeAny(Collection<? extends Callable<T>> tasks)`	Executes given tasks and waits for at least one to complete.

Waiting for Results

```
Future<?> future = service.submit(() →
System.out.println("Hello"));
```

The `Future` type is actually an interface. For the exam, you don't need to know any of the classes that implement `Future`, just that a `Future` instance is returned by various API methods.

Method name	Description
`boolean isDone()`	Returns `true` if task was completed, threw exception, or was cancelled.
`boolean isCancelled()`	Returns `true` if task was cancelled before it completed normally.
`boolean cancel(boolean mayInterruptIfRunning)`	Attempts to cancel execution of task and returns `true` if it was successfully cancelled or `false` if it could not be cancelled or is complete.
`V get()`	Retrieves result of task, waiting endlessly if it is not yet available.
`V get(long timeout, TimeUnit unit)`	Retrieves result of task, waiting specified amount of time. If result is not ready by time timeout is reached, checked `TimeoutException` will be thrown.

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

public class CheckResults {
   private static int counter = 0;
   public static void main(String[] unused) throws Exception {
      try (var service = Executors.newSingleThreadExecutor()) {
         Future<?> result = service.submit(() -> {
            for (int i = 0; i < 1_000_000; i++) counter++;
         });
         result.get(10, TimeUnit.SECONDS);  // Returns null for
Runnable
         System.out.println("Reached!");
      } catch (TimeoutException e) {
         System.out.println("Not reached in time");
      }
    }
}
```

This code also waits at most 10 seconds, throwing a `TimeoutException` on the call to `result.get()` if the task is not done.

Investigating Callable

Callable

Is a functional interface, similar to `Runnable` except that its `call()` method returns a value and can throw a checked exception.

```
@FunctionalInterface public interface Callable<V> {
   V call() throws Exception;
}
```

When `Callable<V>` is passed to an `ExecutorService` via `submit()`, a `Future<V>` object is returned. Once the task is complete, calling `get()` on the `Future<V>` will return the result of type `V`. This allows you to find out a lot of information about the results of the task.

```
try (var service = Executors.newSingleThreadExecutor()) {
   Future<Integer> result = service.submit(() -> 30 + 11);
   System.out.println(result.get());   // 41
}
```

This implementation is easier to code and understand than if we had used a `Runnable`, some shared object, and an `interrupt()` or timed wait. In essence, that's the spirit of the Concurrency API, giving you the tools to write multithreaded code that is thread-safe, performant, and easy to follow.

Shutting down a Thread Executor

A thread executor creates a *non-daemon* thread on the first task that is executed, so forgetting to do this will result in your application *never terminating*.

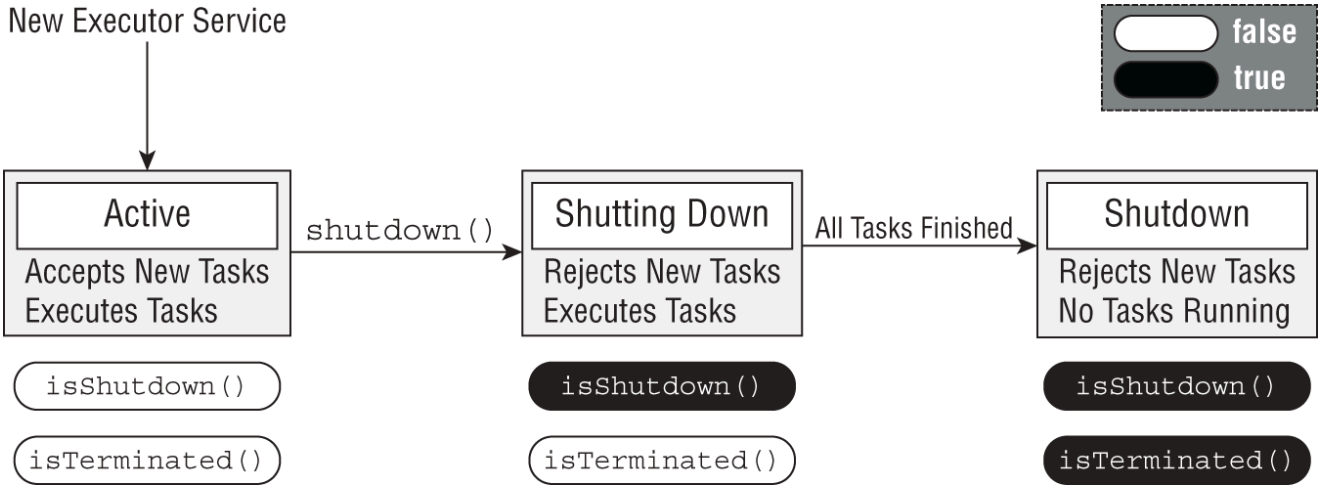
```
public class MissingClose {
   public static void main(String[] args) {
      var service = Executors.newSingleThreadExecutor();
      service.submit(() -> System.out.println("Never stops"));
   } }
```

This code runs but never terminates, because the `ExecutorService` is never shut down or closed. The fix is to always use an `ExecutorService` with a try-with-resources block. The try-with-resources block automatically calls `close()`, which shuts down the executor service so no more tasks get accepted. It then waits until they all complete execution.

TABLE 13.6 `ExecutorService` states

Scenario	Description	More tasks allowed	isShutdown()	isTerminated()
Active	Accepts tasks	Yes	false	false
shutdown()	Runs waiting tasks to completion, but doesn't accept more	No	true	false while tasks running true when tasks complete
close()	Calls shutdown() and then awaits termination of executing tasks	No	true	true
shutdownNow()	Stops executing tasks and cancels waiting tasks	No	true	false while tasks running true when tasks complete

Scheduling Tasks



`ScheduledExecutorService`, which is a subinterface of `ExecutorService`, can be used for just such a task.

Like `ExecutorService`, we obtain an instance of `ScheduledExecutorService` using a factory method in the `Executors` class, as shown in the following snippet:

```
ScheduledExecutorService service =
Executors.newSingleThreadScheduledExecutor();
```

TABLE 13.7 `ScheduledExecutorService` methods

Method name	Description
schedule(Callable<V> callable, long delay, TimeUnit unit)	Creates and executes `Callable` task after given delay
schedule(Runnable command, long delay, TimeUnit unit)	Creates and executes `Runnable` task after given delay
scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)	Creates and executes `Runnable` task after given initial delay, creating new task every period value that passes
scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)	Creates and executes `Runnable` task after given initial delay and subsequently with given delay between termination of one execution and commencement of next

In practice, these methods are among the most convenient in the Concurrency API, as they perform relatively complex tasks with a single line of code. The delay and period parameters rely on the `TimeUnit` argument to determine the format of the value, such as seconds or milliseconds.

```
try (var service = Executors.newSingleThreadScheduledExecutor())
{
   Runnable task1 = () -> System.out.println("Hello Zoo");
   Callable<String> task2 = () -> "Monkey";
   ScheduledFuture<?> r1 = service.schedule(task1, 10,
TimeUnit.SECONDS);
   ScheduledFuture<?> r2 = service.schedule(task2, 8,
TimeUnit.MINUTES);
}
//The first task is scheduled 10 seconds in the future, whereas
the second task is scheduled 8 minutes in the future.
```

The `scheduleAtFixedRate()` method creates a new task and submits it to the executor every period, regardless of whether the previous task finished. The following example executes a `Runnable` task every minute, following an initial five-minute delay:

```
service.scheduleAtFixedRate(task1, 5, 1, TimeUnit.MINUTES);
```

On the other hand, the `scheduleWithFixedDelay()` method creates a new task only after the previous task has finished. For example, if a task runs

at 12:00 and takes five minutes to finish, with a period between executions of two minutes, the next task will start at 12:07.

```
service.scheduleWithFixedDelay(task1, 0, 2, TimeUnit.MINUTES);
```

Increasing Concurrency with Pools

> A *thread pool* is a group of pre-instantiated reusable threads that are available to perform a set of arbitrary tasks.

TABLE 13.8 `Executors` factory methods

Method name	Description
`ExecutorService` **`newSingleThreadExecutor()`**	Creates single-threaded executor that uses single worker platform thread operating off unbounded queue. Results are processed sequentially in the order in which they are submitted.
`ScheduledExecutorService` **`newSingleThreadScheduledExecutor()`**	Creates single-threaded executor for platform threads that can schedule commands to run after given delay or to execute periodically.
`ExecutorService` **`newCachedThreadPool()`**	Creates platform thread pool that creates new threads as needed but reuses previously constructed threads when they are available.
`ExecutorService` **`newFixedThreadPool(int)`**	Creates platform thread pool that reuses fixed number of threads operating off shared unbounded queue.
`ScheduledExecutorService` **`newScheduledThreadPool(int)`**	Creates platform thread pool that can schedule commands to run after given delay or execute periodically.
`ExecutorService` **`newVirtualThreadPerTaskExecutor()`**	Creates thread pool that creates a new virtual thread for each task.

The difference between a single-thread and a pooled-thread executor is what happens when a task is already running. While a single-thread executor will wait for the thread to become available before running the next task, a

pooled-thread executor can execute the next task concurrently. If the pool runs out of available threads, the task will be queued by the thread executor and wait to be completed.

Writing Thread-Safe Code

> *Thread-safety* is the property of an object that guarantees safe execution by multiple threads at the same time.

Understanding Thread-Safety

```
1:  import java.util.concurrent.*;
2:  public class SheepManager {
3:     private int sheepCount = 0;
4:     private void incrementAndReport() {
5:        System.out.print((++sheepCount) + " ");
6:     }
7:     public static void main(String[] args) {
8:        try (var service = Executors.newFixedThreadPool(20)) {
9:           SheepManager manager = new SheepManager();
10:          for (int i = 0; i < 10; i++)
11:             service.submit(() →
manager.incrementAndReport());
12:    } } }
```

It may output in a different order. Worse yet, it may print some numbers twice and not print some numbers at all!

In this example, we use the pre-increment (`++`) operator to update the `sheepCount` variable. A problem occurs when two threads both read the "old" value before either thread writes the "new" value of the variable. The two assignments become redundant; they both assign the same new value, with one thread overwriting the results of the other

Therefore, the increment operator `++` is not thread-safe. As you will see later in this chapter, the unexpected result of two or more tasks executing at the same time is referred to as a *race condition*.

The `volatile` modifier can be used on an instance variable to ensure a thread does not see any intermediary values while an operation is being performed.

Protecting Data with Atomic Classes

> *Atomic* is the property of an operation to be carried out as a single unit of execution without any interference from another thread. A thread-safe atomic version of the increment operator would perform the read and write of the variable as a single operation, not allowing any other threads to access the variable during the operation.

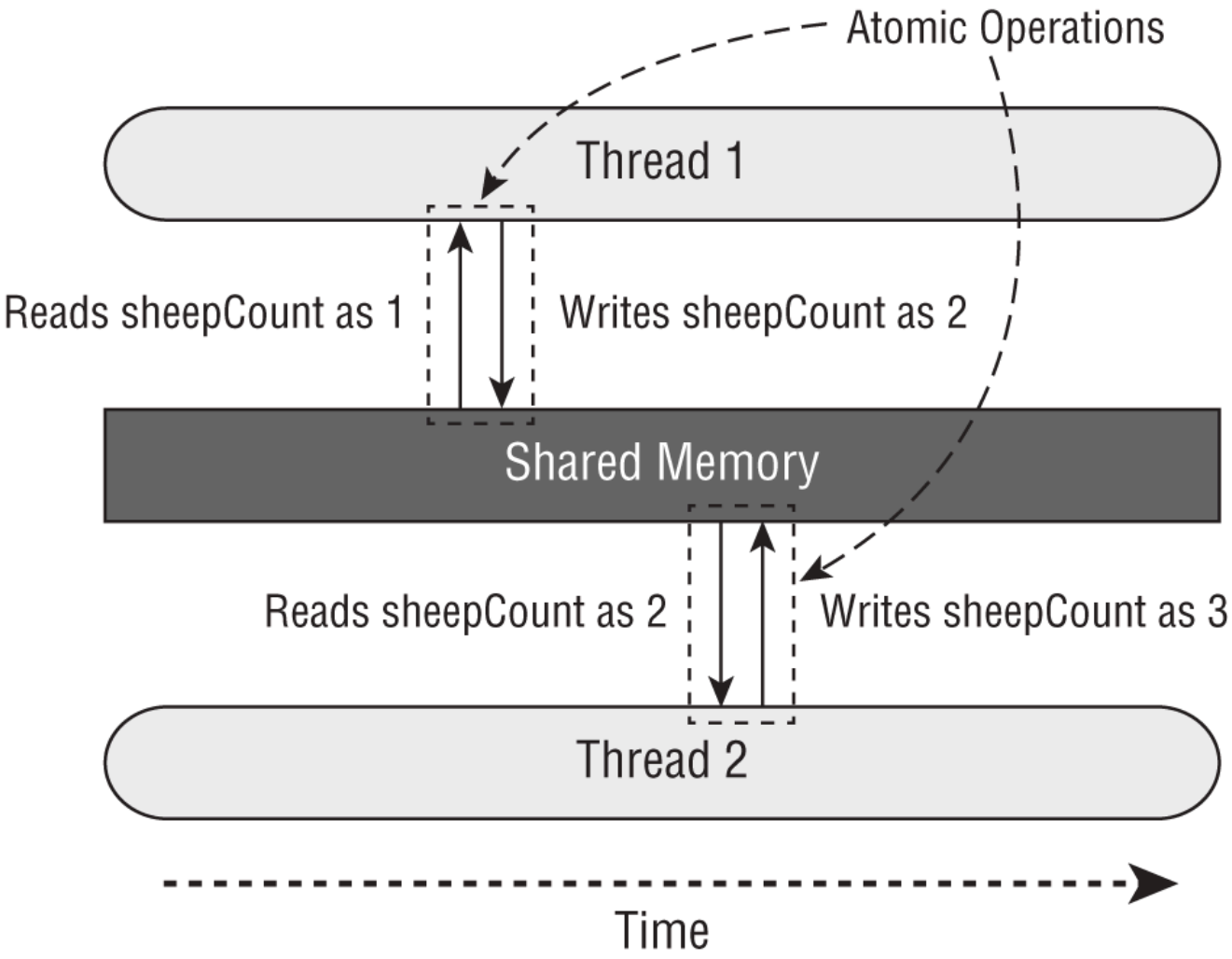


TABLE 13.9 Atomic classes

Class name	Description
AtomicBoolean	A boolean value that may be updated atomically
AtomicInteger	An int value that may be updated atomically
AtomicLong	A long value that may be updated atomically

TABLE 13.10 Common atomic methods

Method name	Description
`get()`	Retrieves current value
`set(`*type* `newValue)`	Sets given value, equivalent to assignment `=` operator
`getAndSet(` *type* `newValue)`	Atomically sets new value and returns old value
`incrementAndGet()`	For numeric classes, atomic pre-increment operation equivalent to `++value`
`getAndIncrement()`	For numeric classes, atomic post-increment operation equivalent to `value++`
`decrementAndGet()`	For numeric classes, atomic pre-decrement operation equivalent to `--value`
`getAndDecrement()`	For numeric classes, atomic post-decrement operation equivalent to `value--`

```
3:    private AtomicInteger sheepCount = new AtomicInteger(0);
4:    private void incrementAndReport() {
5:       System.out.print(sheepCount.incrementAndGet() + " ");
6:    }
//-----------
2 3 1 4 5 6 7 8 9 10
1 4 3 2 5 6 7 8 9 10
1 4 3 5 6 2 7 8 10 9
```

Improving Access with *synchronized* Blocks

> A *monitor*, also called a *lock*, is a structure that supports *mutual exclusion*, which is the property that at most one thread is executing a particular segment of code at a given time.

In Java, any `Object` can be used as a monitor, along with the `synchronized` keyword, as shown in the following example:

```
var manager = new SheepManager();
synchronized(manager) {
   // Work to be completed by one thread at a time
}
```

This is a corrected version of the `SheepManager` class that orders the workers:

```
1:  import java.util.concurrent.*;
2:  public class SheepManager {
3:     private int sheepCount = 0;
4:     private void incrementAndReport() {
5:        synchronized(this) {
6:           System.out.print((++sheepCount) + " ");
7:        }
8:     }
9:     public static void main(String[] args) {
10:       try (var service = Executors.newFixedThreadPool(20)) {
11:          var manager = new SheepManager();
12:          for (int i = 0; i < 10; i++)
13:             service.submit(() →
manager.incrementAndReport());
14:       } } }
```

Synchronizing Methods

We can add the `synchronized` modifier to any instance method to synchronize automatically on the object itself. For example, the following two method definitions are equivalent:

```
void sing() {
 synchronized(this) {
      System.out.print("La la la!");
   }
}
synchronized void sing() {
   System.out.print("La la la!");
}
```

We can also apply the synchronized modifier to static methods. What object is used as the monitor when we synchronize on a static method? The class

object, of course! For example, the following two methods are equivalent for static synchronization inside our SheepManager class:

```
static void dance() {
   synchronized(SheepManager.class) {
      System.out.print("Time to dance!");
   }
}
static synchronized void dance() {
   System.out.print("Time to dance!");
}
```

Understanding the Lock Framework

A `synchronized` block supports only a limited set of functionality. The Concurrency API includes the `Lock` interface, which is conceptually similar to using the `synchronized` keyword but with a lot more bells and whistles.

Applying a *ReentrantLock*

> The `Lock` interface is pretty easy to use. When you need to protect a piece of code from multithreaded processing, create an instance of `Lock` that all threads have access to. Each thread then calls `lock()` before it enters the protected code and calls `unlock()` before it exits the protected code.

```
// Implementation #1 with a synchronized block
var object = new Object();
synchronized(object) {
   // Protected code
}

// Implementation #2 with a Lock
var myLock = new ReentrantLock();
try {
   myLock.lock();
   // Protected code
} finally {
   myLock.unlock();
}
```

While certainly not required, it is a good practice to use a `try` / `finally` block with `Lock` instances to ensure that any acquired locks are properly released. This can help prevent a resource leak in practice.

Besides always making sure to release a lock, you also need to be sure that you only release a lock that you have. If you attempt to release a lock that you do not have, you will get an exception at runtime.

```
var lock = new ReentrantLock();
lock.unlock();  // IllegalMonitorStateException
```

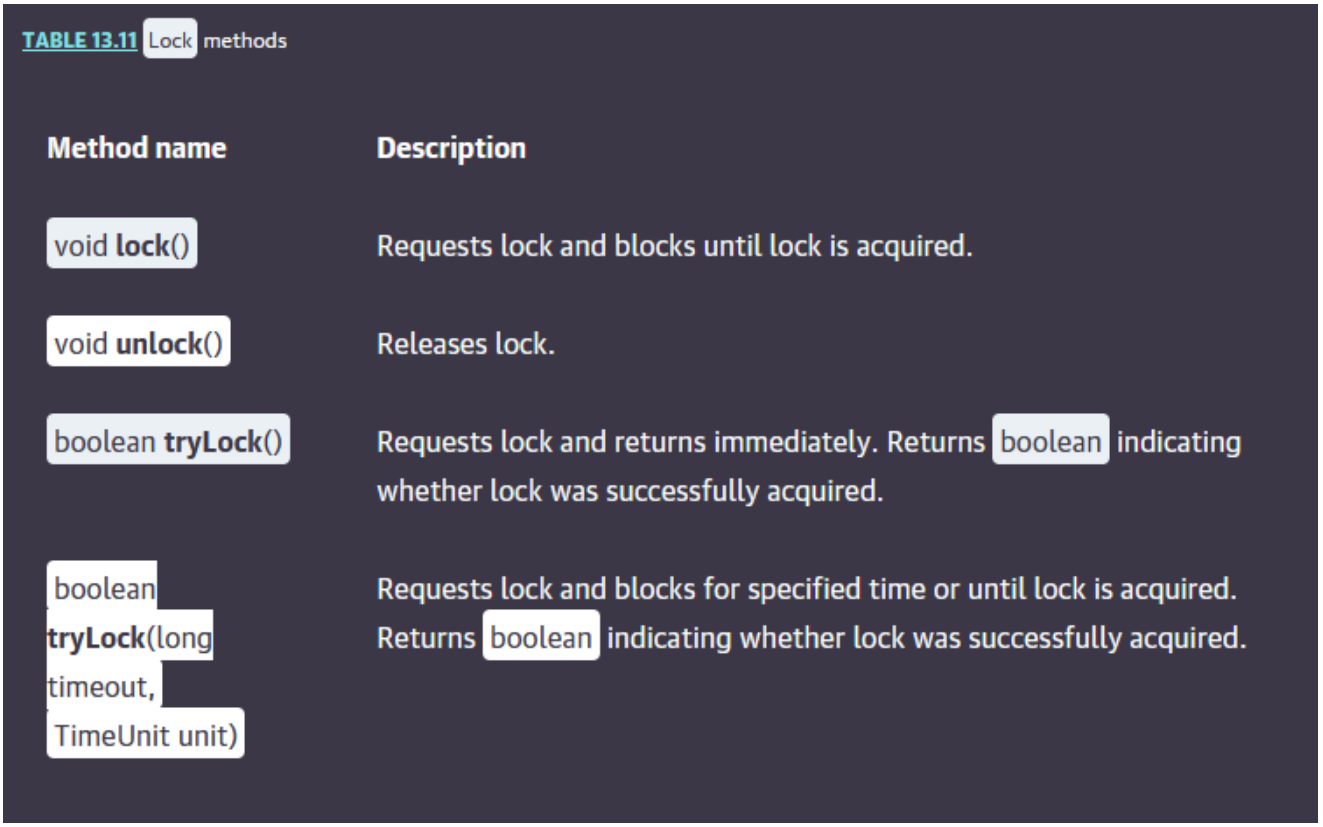


TABLE 13.11 `Lock` methods

Method name	Description
`void lock()`	Requests lock and blocks until lock is acquired.
`void unlock()`	Releases lock.
`boolean tryLock()`	Requests lock and returns immediately. Returns `boolean` indicating whether lock was successfully acquired.
`boolean tryLock(long timeout, TimeUnit unit)`	Requests lock and blocks for specified time or until lock is acquired. Returns `boolean` indicating whether lock was successfully acquired.

Attempting to Acquire a Lock

```
static void printHello(Lock myLock) {
   try {
      myLock.lock();
      System.out.println("Hello");
   } finally {
      myLock.unlock();
   } }
```

The `tryLock()` method will attempt to acquire a lock and immediately return a `boolean` result indicating whether the lock was obtained. Unlike the `lock()` method, it does not wait if another thread already holds the lock. It returns immediately, regardless of whether a lock is available.

```
var myLock = new ReentrantLock();
Thread.ofPlatform().start(() → printHello(myLock));
if (myLock.tryLock()) {
   try {
      System.out.println("Lock obtained, entering protected
code");
   } finally {
      myLock.unlock();
   }
} else {
   System.out.println("Unable to acquire lock, doing something
else");
}
```

When you run this code, it could produce either the `if` or `else` message, depending on the order of execution. It will always print `Hello`, though, as the call to `lock()` in `printHello()` will wait indefinitely for the lock to become available. A fun exercise is to insert some `Thread.sleep()` delays into this snippet to encourage a particular message to be displayed.

The `Lock` interface includes an overloaded version of `tryLock(long,TimeUnit)` that acts like a hybrid of `lock()` and `tryLock()`. Like the other two methods, if a lock is available, it will immediately return with it. If a lock is unavailable, though, it will wait up to the specified time limit for the lock.

Acquiring the Same Lock Twice

```
var myLock = new ReentrantLock();
if (myLock.tryLock()) {
   try {
      myLock.lock();
      System.out.println("Lock obtained, entering protected
code");
   } finally {
      myLock.unlock();
   } }
```

The thread obtains the lock twice but releases it only once.

It is critical that you release a lock the same number of times it is acquired! For calls with `tryLock()`, you need to call `unlock()` only if the method returned `true`.

Reviewing the *Lock* Framework

To review, the `ReentrantLock` class supports the same features as a `synchronized` block while adding a number of improvements:

- Ability to request a lock without blocking.
- Ability to request a lock while blocking for a specified amount of time.
- A lock can be created with a `fairness` property, in which the lock is granted to threads in the order in which it was requested.

Introducing *ReentrantReadWriteLock*

`ReentrantReadWriteLock` is a really useful class. It includes separate locks for reading and writing data. At runtime, only one thread can hold the write lock at a time, but many threads can hold the read lock. Having separate locks can help you maximize concurrent access.

```
var lock = new ReentrantReadWriteLock();
lock.writeLock().lock();

lock.readLock().lock();
System.out.println(lock.isWriteLocked());     // true
System.out.println(lock.getReadLockCount());  // 1

lock.writeLock().unlock();
System.out.println(lock.isWriteLocked());     // false
System.out.println(lock.getReadLockCount());  // 1

lock.readLock().unlock();
System.out.println(lock.getReadLockCount());  // 0
```

On the exam, you are likely to see it used within a single class. You need to know that you can trivially get a read lock after acquiring a write lock because reading is a subset of write. However, if you attempt to get the read lock first, the code will hang as you can't upgrade to a write lock.

Orchestrating Tasks with a *CyclicBarrier*

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

public class LionPenManager {
   private void removeLions() { System.out.println("Removing
lions");   }
```

```
   private void cleanPen()    { System.out.println("Cleaning the
pen"); }
   private void addLions()    { System.out.println("Adding
lions");     }
   public void performTask() {
      removeLions();
      cleanPen();
      addLions();
   }

   public static void main(String[] args) {
      try (var service = Executors.newFixedThreadPool(4)) {
         var manager = new LionPenManager();
         for (int i = 0; i < 4; i++)
            service.submit(() -> manager.performTask());
      } } }

----------
Removing lions
Removing lions
Cleaning the pen
Adding lions
Removing lions
Cleaning the pen
Adding lions
Removing lions
Cleaning the pen
Adding lions
Cleaning the pen
Adding lions
```

Although the results are ordered within a single thread, the output is entirely random among multiple workers. We see that some lions are still being removed while the cage is being cleaned, and other lions are added before the cleaning process is finished.

The `CyclicBarrier` class takes in its constructors a `limit` value, indicating the number of threads to wait for. As each thread finishes, it calls the `await()` method on the cyclic barrier. Once the specified number of threads have each called `await()`, the barrier is released, and all threads can continue.

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

public class LionPenManager {
   private void removeLions() { System.out.println("Removing
lions");   }
   private void cleanPen()    { System.out.println("Cleaning the
pen"); }
   private void addLions()    { System.out.println("Adding
lions");     }
   public void performTask(CyclicBarrier c1, CyclicBarrier c2) {
      try {
         removeLions();
         c1.await();
         cleanPen();
         c2.await();
         addLions();
      } catch (InterruptedException | BrokenBarrierException e) {
         // Handle checked exceptions here
      }
   }

   public static void main(String[] args) {
      try (var service = Executors.newFixedThreadPool(4)) {
         var manager = new LionPenManager();
         var c1 = new CyclicBarrier(4);
         var c2 = new CyclicBarrier(4,
            () -> System.out.println("*** Pen Cleaned!"));
         for (int i = 0; i < 4; i++)
            service.submit(() -> manager.performTask(c1, c2));
      } } }

-------
Removing lions
Removing lions
Removing lions
Removing lions
Cleaning the pen
Cleaning the pen
Cleaning the pen
Cleaning the pen
*** Pen Cleaned!
Adding lions
Adding lions
```

```
Adding lions
Adding lions
```

The `CyclicBarrier` class allows us to perform complex, multithreaded tasks while all threads stop and wait at logical barriers. This solution is superior to a single-threaded solution, as the individual tasks, such as removing the lions, can be completed in parallel by all four zoo workers.

Using Concurrent Collections

Understanding Memory Consistency Errors

> A *memory consistency error* occurs when two threads have inconsistent views of what should be the same data.

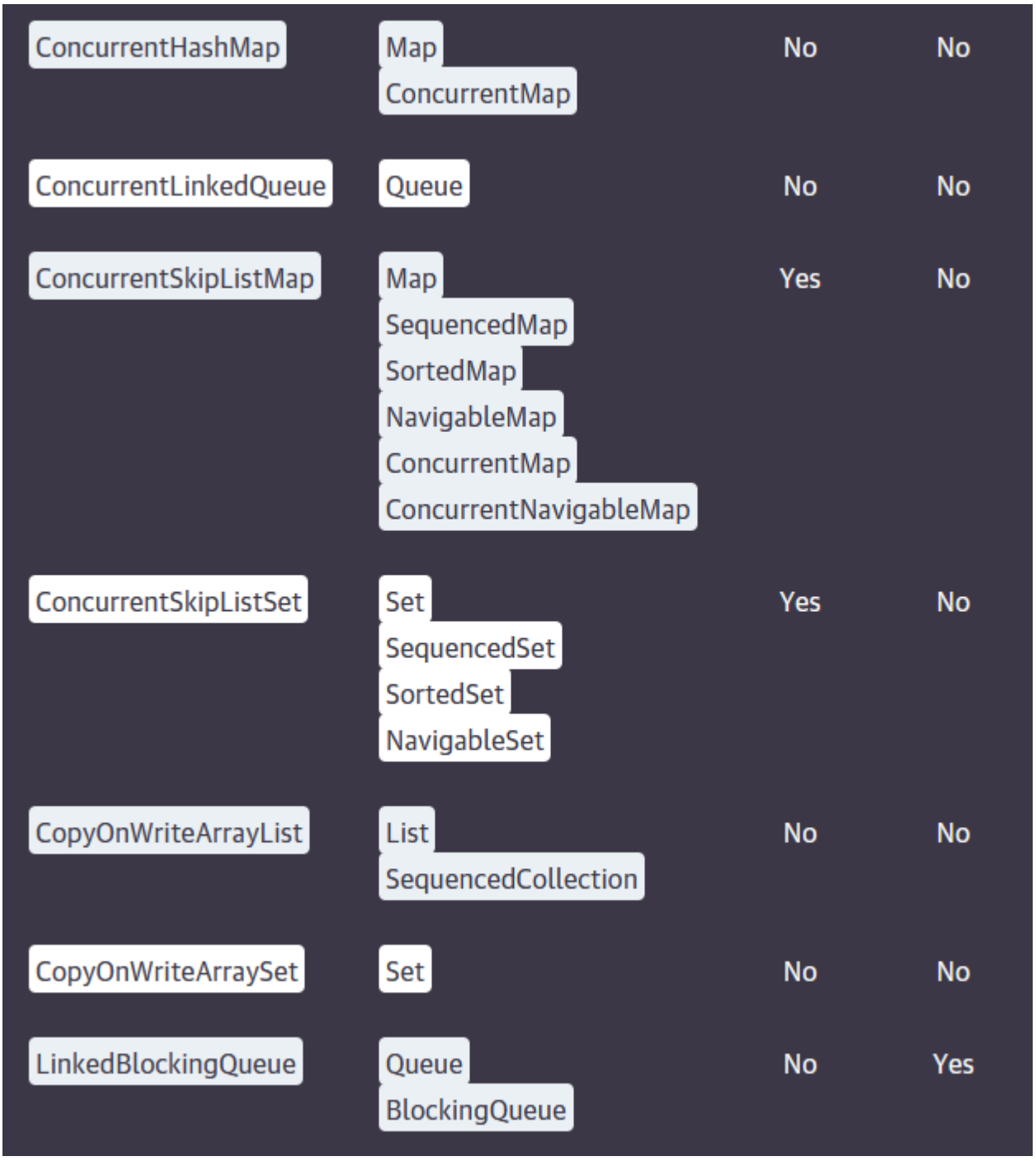
When two threads try to modify the same nonconcurrent collection, the JVM may throw a `ConcurrentModificationException` at runtime.

```
11: var foodData = new HashMap<String, Integer>();
12: foodData.put("penguin", 1);
13: foodData.put("flamingo", 2);
14: for (String key : foodData.keySet())
15:    foodData.remove(key);
```

This snippet will throw a `ConcurrentModificationException` during the second iteration of the loop, since the iterator on `keySet()` is not properly updated after the first element is removed.
Changing the first line to use a `ConcurrentHashMap` will prevent the code from throwing an exception at runtime.

```
11: var foodData = new ConcurrentHashMap<String, Integer>();
```

Working with Concurrent Classes



The `Skip` classes might sound strange, but they are just "sorted" versions of the associated concurrent collections. When you see a class with `Skip` in the name, just think "sorted concurrent" collections, and the rest should follow naturally.

The `CopyOnWrite` classes behave a little differently than the other concurrent data structures you have seen. These classes create a copy of the collection any time a reference is added, removed, or changed in the collection and then update the original collection reference to point to the copy. These classes are commonly used to ensure an iterator doesn't see modifications to the collection.

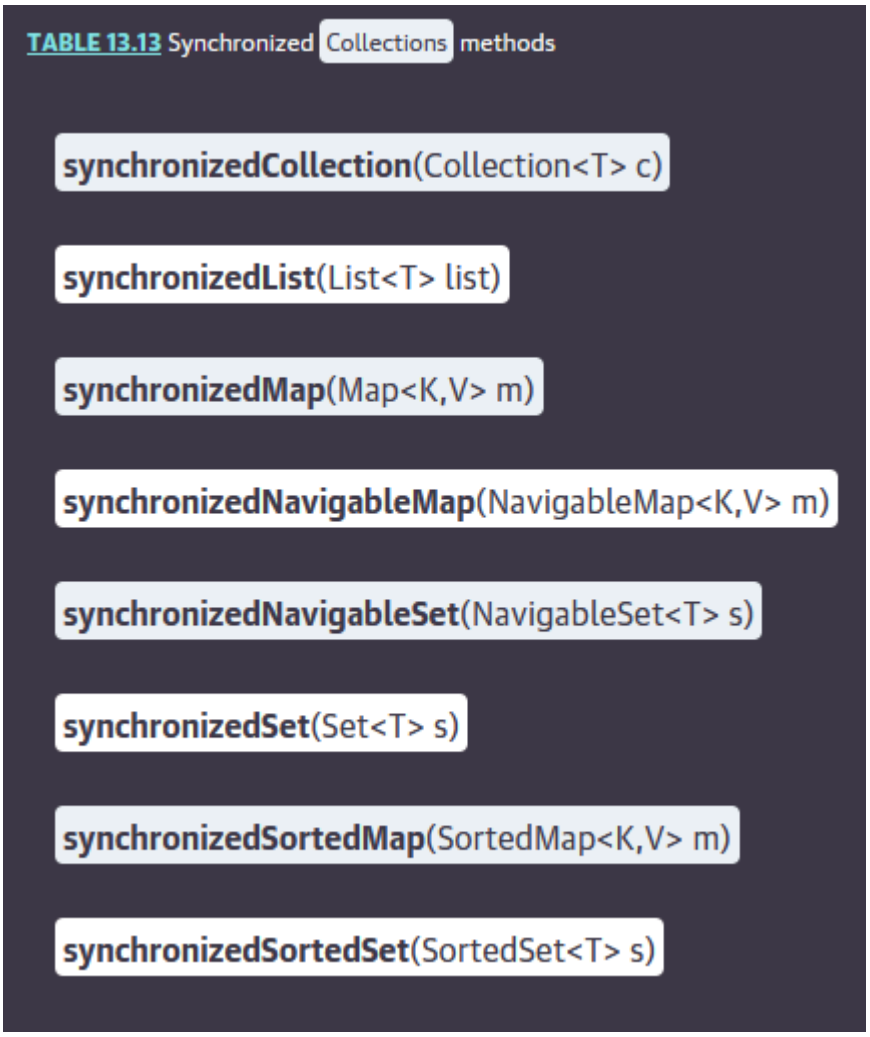
```
List<Integer> favNumbers = new CopyOnWriteArrayList<>(List.of(4,
3, 42));
```

```
for (var n : favNumbers) {
   System.out.print(n + " ");  // 4 3 42
   favNumbers.add(n + 1);
}

System.out.println();
System.out.println("Size: " + favNumbers.size());  // Size: 6
```

`LinkedBlockingQueue`, which implements the concurrent `BlockingQueue` interface. This class is just like a regular `Queue`, except that it includes overloaded versions of `offer()` and `poll()` that take a timeout. These methods wait (or block) up to a specific amount of time to complete an operation.

Obtaining Synchronized Collections



Identifying Threading Problems

Understanding Liveness

Liveness is the ability of an application to be able to execute in a timely manner. Liveness problems, then, are those in which the application

becomes unresponsive or is in some kind of "stuck" state. More precisely, liveness problems are often the result of a thread entering a `BLOCKING` or `WAITING` state forever, or repeatedly entering/exiting these states. For the exam, there are three types of liveness issues with which you should be familiar: deadlock, starvation, and livelock.

Deadlock

Deadlock occurs when two or more threads are blocked forever, each waiting on the other. We can illustrate this principle with the following example. Imagine that our zoo has two foxes: Foxy and Tails. Foxy likes to eat first and then drink water, while Tails likes to drink water first and then eat. Furthermore, neither animal likes to share, and they will finish their meal only if they have exclusive access to both food and water.

Starvation

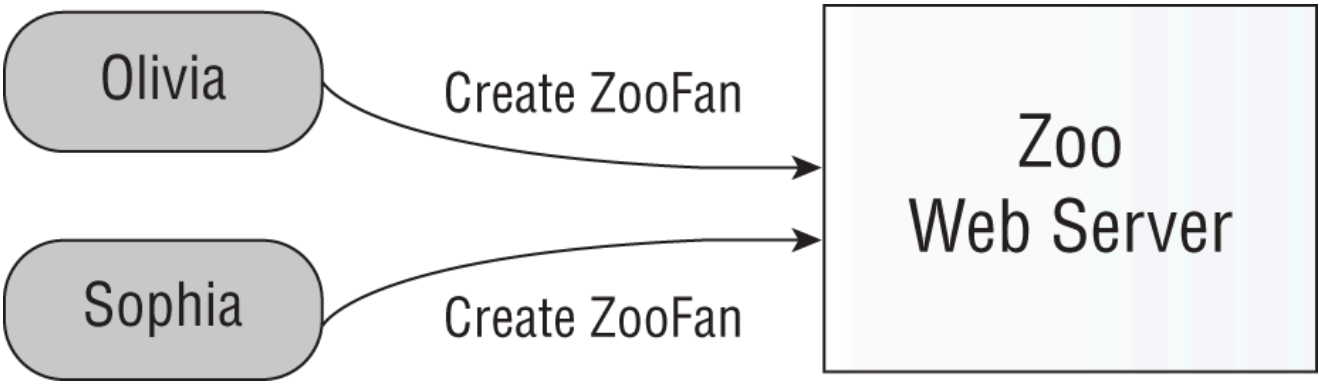
Starvation occurs when a single thread is perpetually denied access to a shared resource or lock. The thread is still active, but it is unable to complete its work as a result of other threads constantly taking the resource that it is trying to access.

Livelock

Livelock occurs when two or more threads are conceptually blocked forever, although they are each still active and trying to complete their task. Livelock is a special case of resource starvation in which two or more threads actively try to acquire a set of locks, are unable to do so, and restart part of the process.

Managing Race Conditions

A *race condition* is an undesirable result that occurs when two tasks that should be completed sequentially are completed at the same time. We encountered examples of race conditions earlier in the chapter when we introduced synchronization.



Possible Outcomes for This Race Condition

- Both users are able to create accounts with the username `ZooFan`.
- Neither user is able to create an account with the username `ZooFan`, and an error message is returned to both users.
- One user is able to create an account with the username `ZooFan`, while the other user receives an error message.
  The first outcome is *really bad*, as it leads to users trying to log in with the same username. Whose data do they see when they log in? The second outcome causes both users to have to try again, which is frustrating but at least doesn't lead to corrupt or bad data.

The third outcome is often considered the best solution. Like the second situation, we preserve data integrity; but unlike the second situation, at least one user is able to move forward on the first request, avoiding additional race condition scenarios.

Working with Parallel Streams

A *parallel stream* is capable of processing results concurrently, using multiple threads. For example, you can use a parallel stream and the `map()` operation to operate concurrently on the elements in the stream, vastly improving performance over processing a single element at a time.

Creating Parallel Streams

```
Collection<Integer> collection = List.of(1, 2);

Stream<Integer> p1 = collection.stream().parallel();
Stream<Integer> p2 = collection.parallelStream();
```

The `Stream` interface includes a method `isParallel()` that can be used to test whether the instance of a stream supports parallel processing. Some operations on streams preserve the parallel attribute, while others do not.

Performing a Parallel Decomposition

A *parallel decomposition* is the process of taking a task, breaking it into smaller pieces that can be performed concurrently, and then reassembling the results. The more concurrent a decomposition, the greater the performance improvement of using parallel streams.

```
private static int doWork(int input) {
   try {
      Thread.sleep(5000);
   } catch (InterruptedException e) {}
```

```
    return input;
}
```

```
10: long start = System.currentTimeMillis();
11: List.of(1, 2, 3, 4, 5)
12:     .stream()
13:     .map(w -> doWork(w))
14:     .forEach(s -> System.out.print(s + " "));
15:
16: System.out.println();
17: var timeTaken = (System.currentTimeMillis()-start)/1000;
18: System.out.println("Time: " + timeTaken + " seconds");
---------------
1 2 3 4 5
Time: 25 seconds
```

```
10: long start = System.currentTimeMillis();
11: List.of(1, 2, 3, 4, 5)
12:     .parallelStream()
13:     .map(w -> doWork(w))
14:     .forEach(s -> System.out.print(s + " "));
15:
16: System.out.println();
17: var timeTaken = (System.currentTimeMillis()-start)/1000;
18: System.out.println("Time: " + timeTaken + " seconds");
---------------
3 2 1 5 4
Time: 5 seconds
```

As you can see, the results are no longer ordered or predictable. The `map()` and `forEach()` operations on a parallel stream are equivalent to submitting multiple `Runnable` lambda expressions to a pooled thread executor and then waiting for the results.
If your stream operation needs to guarantee ordering and you're not sure if it is serial or parallel, you can replace line 14 with one that uses `forEachOrdered()`:

```
14:     .forEachOrdered(s -> System.out.print(s + " "));
```

Processing Parallel Reductions

A *parallel reduction* is a reduction operation applied to a parallel stream. The results for parallel reductions can differ from what you expect when working with serial streams.

Performing Order-Based Tasks

Since order is not guaranteed with parallel streams, methods such as `findAny()` on parallel streams may result in unexpected behavior. What about operations that consider order, such as `findFirst()`, `limit()`, and `skip()`? Order is still preserved, but performance may *suffer* on a parallel stream as a result of a parallel processing task being forced to coordinate all of its threads in a synchronized-like fashion.

It's possible to sort a parallel stream, although the results might not be what you expect. The following prints the numbers from 1 to 99 in a stochastic, or random, ordering:

```
IntStream.range(1,100).parallel().sorted().forEach(System.out::println);
```

After the call to `sorted()`, the stream is still considered parallel, resulting in the `forEach()` method printing the values in a stochastic ordering. Remember to use an ordered method such as `forEachOrdered()` if you need to guarantee ordering on a parallel stream.

On the plus side, the results of ordered operations on a parallel stream will be consistent with a serial stream. For example, calling `skip(5).limit(2).findFirst()` will return the same result on ordered serial and parallel streams.

Combining Results with *reduce()*

Recall that the first parameter to the `reduce()` method is called the *identity*, the second parameter is called the *accumulator*, and the third parameter is called the *combiner*. The following is the signature for the method:

```
<U> U reduce(U identity,
```

We can concatenate a list of `char` values using the `reduce()` method, as shown in the following example:

```
System.out.println(List.of('w', 'o', 'l', 'f')
   .parallelStream()
```

```
    .reduce("",
       (s1, c) -> s1 + c,
       (s2, s3) -> s2 + s3));  // wolf
```

On parallel streams, the `reduce()` method works by applying the reduction to pairs of elements within the stream to create intermediate values and then combining those intermediate values to produce a final result. Put another way, in a serial stream, `wolf` is built one character at a time. In a parallel stream, the intermediate values `wo` and `lf` are created and then combined.

With parallel streams, we now have to be concerned about order. What if the elements of a string are combined in the wrong order to produce `wlfo` or `flwo`? The Stream API prevents this problem while still allowing streams to be processed in parallel, as long as you follow one simple rule: make sure that the accumulator and combiner produce the same result regardless of the order they are called in.

Let's take a look at an example using a problematic accumulator. In particular, order matters when subtracting numbers; therefore, the following code can output different values depending on whether you use a serial or parallel stream. We can omit a combiner parameter in these examples, as the accumulator can be used when the intermediate data types are the same.

```
System.out.println(List.of(1, 2, 3, 4, 5, 6)
```

It may output `-21`, `3`, or some other value.

You can see other problems if we use an identity parameter that is not truly an identity value. For example, what do you expect the following code to output?

```
System.out.println(List.of("w","o","l","f")
```

On a serial stream, it prints `Xwolf`, but on a parallel stream, the result is `XwXoXlXf`. As part of the parallel process, the identity is applied to multiple elements in the stream, resulting in very unexpected data.

Combining Results with *collect()*

Like reduce(), the Stream API includes a three-argument version of collect() that takes accumulator and combiner operators along with a supplier operator instead of an identity.

```
<R> R collect(Supplier<R> supplier,
   BiConsumer<R, ? super T> accumulator,
   BiConsumer<R, R> combiner)
```

Also, like reduce(), the accumulator and combiner operations must be able to process results in any order. In this manner, the three-argument version of collect() can be performed as a parallel reduction, as shown in the following example:

```
Stream<String> stream = Stream.of("w", "o", "l", "f").parallel();
SortedSet<String> set =
stream.collect(ConcurrentSkipListSet::new,
   Set::add,
   Set::addAll);
System.out.println(set);  // [f, l, o, w]
```

Recall that elements in a `ConcurrentSkipListSet` are sorted according to their natural ordering. You should use a concurrent collection to combine the results, ensuring that the results of concurrent threads do not cause a `ConcurrentModificationException`.

Performing parallel reductions with a collector requires additional considerations. For example, if the collection into which you are inserting is an ordered data set, such as a `List`, the elements in the resulting collection must be in the same order, regardless of whether you use a serial or parallel stream. This may reduce performance, though, as some operations cannot be completed in parallel.

Performing a Parallel Reduction on a Collector

Every `Collector` instance defines a `characteristics()` method that returns a set of `Collector.Characteristics` attributes. When using a `Collector` to perform a parallel reduction, a number of properties must hold true. Otherwise, the `collect()` operation will execute in a single-threaded fashion.

Requirements for Parallel Reduction with *collect()*

- The stream is parallel.
- The parameter of the `collect()` operation has the `Characteristics.CONCURRENT` characteristic.
- Either the stream is unordered or the collector has the characteristic `Characteristics.UNORDERED`.

For example, while `Collectors.toSet()` does have the `UNORDERED` characteristic, it does not have the `CONCURRENT` characteristic. Therefore, the following is not a parallel reduction even with a parallel stream:

```
parallelStream.collect(Collectors.toSet());  // Not a parallel
reduction
```

The `Collectors` class includes two sets of `static` methods for retrieving collectors, `toConcurrentMap()` and `groupingByConcurrent()`, both of which are `UNORDERED` and `CONCURRENT`. These methods produce `Collector` instances capable of performing parallel reductions efficiently. Like their nonconcurrent counterparts, there are overloaded versions that take additional arguments.

```
Stream<String> ohMy = Stream.of("lions", "tigers",
"bears").parallel();
ConcurrentMap<Integer, String> map = ohMy
   .collect(Collectors.toConcurrentMap(String::length,
      k -> k,
      (s1, s2) -> s1 + "," + s2));
System.out.println(map);              // {5=lions,bears, 6=tigers}
System.out.println(map.getClass());  //
java.util.concurrent.ConcurrentHashMap
```

We use a `ConcurrentMap` reference, although the actual class returned is likely `ConcurrentHashMap`. The particular class is not guaranteed; it will just be a class that implements the interface `ConcurrentMap`.

```
var ohMy = Stream.of("lions", "tigers", "bears").parallel();
ConcurrentMap<Integer, List<String>> map = ohMy.collect(
   Collectors.groupingByConcurrent(String::length));
System.out.println(map);              // {5=[lions, bears], 6=
[tigers]}
```

Side effects can appear in parallel streams if your lambda expressions are stateful. A *stateful lambda expression* is one whose result depends on any state that might change during the execution of a pipeline.