11

Concurrency

CERTIFICATION OBJECTIVES

- Create Worker Threads Using Runnable, Callable and Use an ExecutorService to Concurrently Execute Tasks
- Use Synchronized Keyword and java.util.concurrent.atomic Package to Control the Order of Thread Execution
- Use java.util.concurrent Collections and Classes Including CyclicBarrier and CopyOnWriteArrayList
- Use Parallel Fork/Join Framework
- Use Parallel Streams Including Reduction, Decomposition, Merging Processes, Pipelines and Performance
- Identify Potential Threading Problems among Deadlock, Starvation, Livelock, and Race Conditions

✓ Two-Minute Drill

Q&A Self Test

Concurrency with the java.util.concurrent Package

As you learned in the previous chapter on threads, the Java platform supports multithreaded programming. Supporting multithreaded programming is essential for any modern programming language because servers, desktop computers, laptops, and most mobile devices contain multiple CPUs. If you want your applications to take advantage of all of the processing power present in a modern system, you must create multithreaded applications.

Unfortunately, creating efficient and error-free multithreaded applications can be a challenge. The low-level threading constructs such as `Thread`, `Runnable`, `wait()`, `notify()`, and synchronized blocks are too primitive for many requirements and force developers to create their own high-level threading libraries. Custom threading libraries can be both error prone and time consuming to create.

The `java.util.concurrent` package provides high-level APIs that support many common concurrent programming use cases. When possible, you should use these high-

level APIs in place of the traditional low-level threading constructs (synchronized, wait, notify). Some features (such as the locking API) provide functionality similar to what existed already, but with more flexibility at the cost of slightly awkward syntax. Using the `java.util.concurrent` classes requires a solid understanding of the traditional Java threading types (`Thread` and `Runnable`) and their use (start, run, synchronized, wait, notify, join, sleep, etc.). If you are not comfortable with Java threads, you should return to the previous chapter before continuing with these high-level concurrency APIs.

CERTIFICATION OBJECTIVE

Apply Atomic Variables and Locks (OCP Objective 10.3)

10.3 Use synchronized keyword and java.util.concurrent.atomic package to control the order of thread execution.

The `java.util.concurrent.atomic` and `java.util.concurrent.locks` packages solve two different problems. They are grouped into a single exam objective simply because they are the only two packages below `java.util.concurrent` and both have a small number of classes and interfaces to learn. The `java.util.concurrent.atomic` package enables multithreaded applications to safely access individual variables without locking, whereas the `java.util.concurrent.locks` package provides a locking framework that can be used to create locking behaviors that are the same or superior to those of Java's synchronized keyword.

Atomic Variables

Imagine a multiplayer video game that contains monsters that must be destroyed. The players of the game (threads) are vanquishing monsters, while at the same time a monster-spawning thread is repopulating the world to ensure players always have a new challenge to face. To keep the level of difficulty consistent, you would need to keep track of the monster count and ensure that the monster population stays the same (a hero's work is never done). Both the player threads and the monster-spawning thread must access and modify the shared monster count variable. If the monster count somehow became incorrect, your players may find themselves with more adversaries than they could handle.

The following example shows how even the seemingly simplest of code can lead to undefined results. Here you have a class that increments and reports the current value of an integer variable:

```
public class Counter {
  private int count;
  public void increment() {
    count++; // it's a trap!
             // a single "line" is not atomic
  }
  public int getValue() {
    return count;
  }
}
```

A `Thread` that will increment the counter 10,000 times:

```
public class IncrementerThread extends Thread {
  private Counter counter;
  // all instances are passed the same counter
  public IncrementerThread(Counter counter) {
    this.counter = counter;
  }
  public void run() {
    // "i" is local and therefore thread-safe
     for(int i = 0; i < 10000; i++) {
       counter.increment();
     }
   }
}
```

Here is the code from within this application's main method:

```
Counter counter = new Counter();        // the shared object
IncrementerThread it1 = new IncrementerThread(counter);
IncrementerThread it2 = new IncrementerThread(counter);
it1.start(); // thread 1 increments the count by 10000
it2.start(); // thread 2 increments the count by 10000
it1.join();  // wait for thread 1 to finish
it2.join();  // wait for thread 2 to finish
System.out.println(counter.getValue()); // rarely 20000
                                        // lowest 11972
```

The trap in this example is that `count++` looks like a single action when, in fact, it is not. When incrementing a field like this, what *probably* happens is the following sequence:

1. The value stored in `count` is copied to a temporary variable.
2. The temporary variable is incremented.
3. The value of the temporary variable is copied back to the `count` field.

We say "probably" in this example because while the Java compiler will translate the `count++` statement into multiple Java bytecode instructions, you really have no control over what native instructions are executed. The JIT (Just In Time compiler)–based nature of most Java runtime environments means you don't know when or if the `count++` statement will be translated to native CPU instructions and whether it ends up as a single instruction or several. You should always act as if a single line of Java code takes multiple steps to complete. Getting an incorrect result also depends on many other factors, such as the type of CPU you have. Do both threads in the example run concurrently or in sequence? A large loop count was used in order to make the threads run longer and be more likely to execute concurrently.

While you could make this code thread-safe with synchronized blocks, the act of obtaining and releasing a lock flag would probably be more time consuming than the work being performed. This is where the classes in the `java.util.concurrent.atomic` package can benefit you. They provide variables whose values can be modified atomically. An atomic operation is one that, for all intents and purposes, appears to happen all at once. The `java.util.concurrent.atomic` package provides several classes for different data types, such as `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, and `AtomicReference`, to name a few.

Here is a thread-safe replacement for the `Counter` class from the previous example:

```
public class Counter {
  private AtomicInteger count = new AtomicInteger();
  public void increment() {
    count.getAndIncrement(); // atomic operation
  }
  public int getValue() {
    return count.intValue();
  }
}
```

In reality, even a method such as `getAndIncrement()` still takes several steps to execute. The reason this implementation is now thread-safe is something called CAS. CAS stands for Compare And Swap. Most modern CPUs have a set of CAS instructions. Following is a basic outline of what is happening now:

1. The value stored in `count` is copied to a temporary variable.
2. The temporary variable is incremented.
3. Compare the value currently in `count` with the original value. If it is unchanged, then swap the old value for the new value.

Step 3 happens atomically. If step 3 finds that some other thread has already modified the value of `count`, then repeat steps 1–3 until we increment the field without interference.

The central method in a class like `AtomicInteger` is the `boolean compareAndSet(int expect, int update)` method, which provides the CAS behavior. Other atomic methods delegate to the `compareAndSet` method. The `getAndIncrement` method implementation is simply:

```
public final int getAndIncrement() {
  for (;;) {
    int current = get();
    int next = current + 1;
    if (compareAndSet(current, next))
      return current;
  }
}
```

Locks

The `java.util.concurrent.locks` package is about creating (not surprisingly) locks. Why would you want to use locks when so much of `java.util.concurrent` seems geared toward avoiding overt locking? You use `java.util.concurrent.locks` classes and traditional monitor locking (the synchronized keyword) for roughly the same purpose: creating segments of code that require exclusive execution (one thread at a time).

Why would you create code that limits the number of threads that can execute it? While atomic variables work well for making single variables thread-safe, imagine if you have two or more variables that are related. A video game character might have a number of gold pieces that can be carried in his backpack and a number of gold pieces he keeps in an in-game bank vault. Transferring gold into the bank is as simple as subtracting gold from the backpack and adding it to the vault. If we have 10 gold pieces in our backpack and 90 in the vault, we have a total of 100 pieces that belong to our character. If we want to transfer all 10 pieces to the vault, we can first add 10 to the vault count and then subtract 10 from the backpack, or first subtract 10 from the backpack and then add 10 to the vault. If another thread were to try to assess our character's wealth during the middle of our transfer, it might see 90 pieces or 110 pieces, depending on the order of our operations, neither being the correct count of 100 pieces.

This other thread that is attempting to read the character's total wealth might do all sorts of things, such as increase the likelihood of your character being robbed or a variety of other actions to control the in-game economics. All game threads must correctly gauge a character's wealth even if there is a transfer in progress.

The solution to our balance inquiry transfer problem is to use locking. Create a single method to get a character's wealth and another to perform gold transfers. You should never be able to check a character's total wealth while a gold transfer is in progress. Having a single method to get a character's total wealth is also important because you don't want a thread to read the backpack's gold count before a transfer and then the vault's gold count after a transfer. That would lead to the same incorrect total as trying to calculate the total

during a transfer.

Much of the functionality provided by the classes and interfaces of the `java.util.concurrent.locks` package duplicates that of traditional synchronized locking. In fact, the hypothetical gold transfer outlined earlier could be solved with either the synchronized keyword or classes in the `java.util.concurrent.locks` package. In Java 5, when `java.util.concurrent` was first introduced, the new locking classes performed better than the synchronized keyword, but there is no longer a vast difference in performance. So why would you use these newer locking classes? The `java.util.concurrent.locks` package provides

- The ability to duplicate traditional synchronized blocks.
- Nonblock scoped locking—obtain a lock in one method and release it in another (this can be dangerous, though).
- Multiple `wait`/`notify`/`notifyAll` pools per lock—threads can select which pool (`Condition`) they wait on.
- The ability to attempt to acquire a lock and take an alternative action if locking fails.
- An implementation of a multiple-reader, single-writer lock.

ReentrantLock

The `java.util.concurrent.locks.Lock` interface provides the outline of locking provided by the `java.util.concurrent.locks` package. Like any interface, the `Lock` interface requires an implementation to be of any real use. The `java.util.concurrent.locks.ReentrantLock` class provides that implementation. To demonstrate the use of `Lock`, we will first duplicate the functionality of a basic traditional synchronized block.

```
Object obj = new Object();
synchronized(obj) {   // traditional locking, blocks until acquired
                      // work
}                     // releases lock automatically
```

Here is an equivalent piece of code using the `java.util.concurrent.locks` package. Notice how `ReentrantLock` can be stored in a `Lock` reference because it implements the `Lock` interface. This example blocks on attempting to acquire a lock, just like traditional synchronization.

```
Lock lock = new ReentrantLock();
lock.lock();          // blocks until acquired
try {
    // do work here
} finally {           // to ensure we unlock
  lock.unlock();      // must manually release
}
```

It is recommended that you always follow the `lock()` method with a `try-finally` block, which releases the lock. The previous example doesn't really provide a compelling reason for you to choose to use a `Lock` instance instead of traditional synchronization. One of the very powerful features is the ability to attempt (and fail) to acquire a lock. With traditional synchronization, once you hit a synchronized block, your thread either immediately acquires the lock or blocks until it can.

```
Lock lock = new ReentrantLock();
boolean locked = lock.tryLock(); // try without waiting
if (locked) {
  try {
      // work
  } finally {                    // to ensure we unlock
    lock.unlock();
  }
}
```

The ability to quickly fail to acquire the lock turns out to be powerful. You can process a different resource (lock) and come back to the failed lock later instead of just waiting for a lock to be released and thereby making more efficient use of system resources. There is also a variation of the `tryLock` method that allows you to specify an amount of time you are willing to wait to acquire the lock:

```
Lock lock = new ReentrantLock();
try {
  boolean locked = lock.tryLock(3, TimeUnit.SECONDS);
  if (locked) {
    try {
                        // work
    } finally {         // to ensure we unlock
       lock.unlock();
    }
  }
} catch (InterruptedException ex) {
                        // handle
}
```

Another benefit of the `tryLock` method is deadlock avoidance. With traditional synchronization, you must acquire locks in the same order across all threads. For example, if you have two objects to lock against:

```
Object o1 = new Object();
Object o2 = new Object();
```

and you synchronize using the internal lock flags of both objects:

```
synchronized(o1) {
          // thread A could pause here
  synchronized(o2) {
          // work
  }
}
```

you should never acquire the locks in the opposite order because it could lead to deadlock. Although thread A has only the `o1` lock, thread B acquires the `o2` lock. You are now at an

impasse because neither thread can obtain the second lock it needs to continue.

```
synchronized(o2) {
          // thread B gets stuck here
  synchronized(o1) {
          // work
  }
}
```

Looking at a similar example using a `ReentrantLock`, start by creating two locks:

```
Lock l1 = new ReentrantLock();
Lock l2 = new ReentrantLock();
```

Next, you acquire both locks in thread A:

```
boolean aq1 = l1.tryLock();
boolean aq2 = l2.tryLock();
try{
  if (aq1 && aq2) {
      // work
  }
} finally {
  if (aq2) l2.unlock(); // don't unlock if not locked
  if (aq1) l1.unlock();
}
```

Notice the example is careful to always unlock any acquired lock, but ONLY the lock(s) that were acquired. A `ReentrantLock` has an internal counter that keeps track of the number of times it has been locked/unlocked, and it is an error to unlock without a corresponding successful lock operation. If a thread attempts to release a lock that it does not own, an `IllegalMonitorStateException` will be thrown.

Now in thread B, the locks are obtained in the reverse order in which thread A obtained them. With traditional locking, using synchronized code blocks and attempting to obtain

locks in the reverse order could lead to deadlock.

```
boolean aq2 = l2.tryLock();
boolean aq1 = l1.tryLock();
try{
  if (aq1 && aq2) {
      // work
  }
} finally {
  if (aq1) l1.unlock();
  if (aq2) l2.unlock();
}
```

Now, even if thread A was only in possession of the `l1` lock, there is no possibility that thread B could block because we use the nonblocking `tryLock` method. Using this technique, you can avoid deadlocking scenarios, but you must deal with the possibility that both locks could not be acquired. Using a simple loop, you can repeatedly attempt to obtain both locks until successful (Note: This approach is CPU intensive; we'll look at a better solution later):

```
loop2:
while (true) {
  boolean aq2 = l2.tryLock();
  boolean aq1 = l1.tryLock();
  try {
    if (aq1 && aq2) {
      // work
      break loop2;
    }
  } finally {
    if (aq2) l2.unlock();
    if (aq1) l1.unlock();
  }
}
```

It is remotely possible that this example could lead to livelock. Imagine if thread A always acquires ***`lock1`*** *at the same time that thread B acquires* ***`lock2`****. Each thread's attempt to acquire the second lock would always fail, and you'd end up repeating forever, or at least until you were lucky enough to have one thread fall behind the other. You can avoid livelock in this scenario by introducing a short random delay with* ***`Thread.sleep(int)`*** *any time you fail to acquire both locks.*

Condition

A `Condition` provides the equivalent of the traditional `wait`, `notify`, and `notifyAll` methods. The traditional `wait` and `notify` methods allow developers to implement an await/signal pattern. You use an await/signal pattern when you would use locking, but with the added stipulation of trying to avoid spinning (endless checking if it is okay to do something). Imagine a video game character who wants to buy something from a store, but the store is out of stock at the moment. The character's thread could repeatedly lock the store object and check for the desired item, but that would lead to unneeded system

utilization. Instead, the character's thread can say, "I'm taking a nap, wake me up when new stock arrives."

The `java.util.concurrent.locks.Condition` interface is a replacement for the `wait` and `notify` methods. A three-part code example shows you how to use a condition. Part one shows that a `Condition` is created from a `Lock` object:

```
Lock lock = new ReentrantLock();
Condition blockingPoolA = lock.newCondition();
```

When your thread reaches a point where it must delay until another thread performs an activity, you "await" the completion of that other activity. Before calling `await`, you must have locked the `Lock` used to produce the `Condition`. It is possible that the awaiting thread may be interrupted, and you must handle the possible `InterruptedException`. When you call the `await` method, the `Lock` associated with the `Condition` is released. Before the `await` method returns, the lock will be reacquired. In order to use a `Condition`, a thread must first acquire a `Lock`. Part two of the three-part `Condition` example shows how a `Condition` is used to pause or wait for some event:

```
lock.lock();
try {
  blockingPoolA.await(); // "wait" here
                         // lock will be reacquired
    // work
} catch (InterruptedException ex) {
    // interrupted during await()
} finally {              // to ensure we unlock
  lock.unlock();         // must manually release
}
```

In another thread, you perform the activity that the first thread was waiting on and then signal that first thread to resume (return from the `await` method). Part three of the `Condition` example is run in a different thread than part two. This part causes the thread waiting in the second piece to wake up:

```
lock.lock();
try {
  // work
  blockingPoolA.signalAll(); // wake all awaiting
                             // threads
} finally {
  lock.unlock();             // now an awoken thread can run
}
```

The `signalAll()` method causes all threads awaiting on the same `Condition` to wake up. You can also use the `signal()` method to wake up a single awaiting thread. Remember that "waking up" is not the same thing as proceeding. Each awoken thread will have to reacquire the `Lock` before continuing.

One advantage of a `Condition` over the traditional wait/notify operations is that multiple `Conditions` can exist for each `Lock`. A `Condition` is effectively a waiting/blocking pool for threads.

```
Lock lock = new ReentrantLock();
Condition blockingPoolA = lock.newCondition();
Condition blockingPoolB = lock.newCondition();
```

By having multiple conditions, you are effectively categorizing the threads waiting on a lock and can, therefore, wake up a subset of the waiting threads.

`Conditions` can also be used when you can't use a `BlockingQueue` to coordinate the activities of two or more threads.

ReentrantReadWriteLock

Imagine a video game that was storing a collection of high scores using a non-thread-safe collection. With a non-thread-safe collection, it is important that if a thread is attempting to modify the collection, it must have exclusive access to the collection. To allow multiple threads to concurrently read the high score list or allow a single thread to add a new score, you could use a `ReadWriteLock`.

A `ReentrantReadWriteLock` is not actually a `Lock`; it implements the `ReadWriteLock` interface. What a `ReentrantReadWriteLock` does is produce two specialized `Lock` instances, one to a read lock and the other to a write lock.

```
ReentrantReadWriteLock rwl =
  new ReentrantReadWriteLock();
Lock readLock = rwl.readLock();
Lock writeLock = rwl.writeLock();
```

These two locks are a matched set—one cannot be held at the same time as the other (by different threads). What makes these locks unique is that multiple threads can hold the read lock at the same time, but only one thread can hold the write lock at a time.

This example shows how a non-thread-safe collection (an `ArrayList`) can be made thread-safe, allowing concurrent reads but exclusive access by a writing thread:

```
public class MaxValueCollection {
    private List<Integer> integers = new ArrayList<>();
    private ReentrantReadWriteLock rwl =
        new ReentrantReadWriteLock();
```

```
    public void add(Integer i) {
        rwl.writeLock().lock(); // one at a time
        try {
            integers.add(i);
        } finally {
            rwl.writeLock().unlock();
        }
    }

    public int findMax() {
        rwl.readLock().lock(); // many at once
        try {
            return Collections.max(integers);
        } finally {
            rwl.readLock().unlock();
        }
    }
}
```

Instead of wrapping a collection with `Lock` objects to ensure thread safety, you can use one of the thread-safe collections you'll learn about in the next section.

CERTIFICATION OBJECTIVE

Use java.util.concurrent Collections (OCP Objective 10.4)

10.4 Use java.util.concurrent collections and classes including CyclicBarrier and CopyOnWriteArrayList.

Imagine an online video game with a list of the top 20 scores in the last 30 days. You could model the high score list using a `java.util.ArrayList`. As scores expire, they are removed from the list, and as new scores displace existing scores, remove and insert

operations are performed. At the end of every game, the list of high scores is displayed. If the game is popular, then a lot of people (threads) will be reading the list at the same time. Occasionally, the list will be modified—734sometimes by multiple threads—probably at the same time that it is being read by a large number of threads.

A traditional `java.util.List` implementation such as `java.util.ArrayList` is not thread-safe. Concurrent threads can safely read from an `ArrayList` and possibly even modify the elements stored in the list, but if any thread modifies the structure of the list (add or remove operation), then unpredictable behavior can occur.

Look at the `ArrayListRunnable` class in the following example. What would happen if there were a single instance of this class being executed by several threads? You might encounter several problems, including `ArrayIndexOutOfBoundsException`, duplicate values, skipped values, and null values. Not all threading problems manifest immediately. To observe the bad behavior, you might have to execute the faulty code multiple times or under different system loads. It is important that you are able to recognize the difference between thread-safe and non-thread-safe code yourself, because the compiler will not detect thread-unsafe code.

```
public class ArrayListRunnable implements Runnable {
  // shared by all threads
  private List<Integer> list = new ArrayList<>();

  public ArrayListRunnable() {
    // add some elements
    for (int i = 0; i < 100000; i++) {
      list.add(i);
    }
  }

    // might run concurrently, you cannot be sure
    // to be safe you must assume it does
  public void run() {
    String tName = Thread.currentThread().getName();
    while (!list.isEmpty()) {
      System.out.println(tName + " removed " + list.remove(0));
    }
  }

  public static void main(String[] args) {
    ArrayListRunnable alr = new ArrayListRunnable();
    Thread t1 = new Thread(alr);
    Thread t2 = new Thread(alr); // shared Runnable
    t1.start();
    t2.start();
  }
}
```

To make a collection thread-safe, you could surround all the code that accessed the collection in synchronized blocks or use a method such as `Collections.synchronizedList(new ArrayList())`. Using synchronization to safeguard a collection creates a performance bottleneck and reduces the liveness of your application. The `java.util.concurrent` package provides several types of collections that are thread-safe but do not use coarse-grained synchronization. When a collection will be concurrently accessed in an application you are developing, you should always consider using the collections outlined in the following sections.

Problems in multithreaded applications may not always manifest—a lot depends on the underlying operating system and how other applications affect the thread scheduling of a problematic application. On the exam, you might be asked about the "probable" or "most likely" outcome. Unless you are asked to identify every possible outcome of a code sample, don't get hung up on unlikely results. For example, if a code sample uses ***`Thread.sleep(1000)`*** *and nothing indicates that the thread would be interrupted while it was sleeping, it would be safe to assume that the thread would resume execution around one second after the call to sleep.*

Copy-on-Write Collections

The copy-on-write collections from the `java.util.concurrent` package implement one of several mechanisms to make a collection thread-safe. By using the copy-on-write collections, you eliminate the need to implement synchronization or locking when manipulating a collection using multiple threads.

The `CopyOnWriteArrayList` is a `List` implementation that can be used concurrently without using traditional synchronization semantics. As its name implies, a `CopyOnWriteArrayList` will never modify its internal array of data. Any mutating operations on the `List` (add, set, remove, etc.) will cause a new modified copy of the array to be created, which will replace the original read-only array. The read-only nature of the underlying array in a `CopyOnWriteArrayList` allows it to be safely shared with multiple threads. Remember that read-only (immutable) objects are always thread-safe.

The essential thing to remember with a copy-on-write collection is that a thread that is looping through the elements in a collection must keep a reference to the same unchanging elements throughout the duration of the loop; this is achieved with the use of an `Iterator`. You want to keep using the old, unchanging collection that you began a loop with. When you use `list.iterator()`, the returned `Iterator` will always reference the collection of elements as it was when `list.iterator()` was called, even if another thread

modifies the collection. Any mutating methods called on a copy-on-write–based `Iterator` or `ListIterator` (such as add, set, or remove) will throw an `UnsupportedOperationException`.

A ***for-each*** *loop uses an* ***Iterator*** *when executing, so it is safe to use with a copy-on-write collection, unlike a traditional* ***for*** *loop.*

```
for(Object o : collection) {}                    // use this
for(int i = 0; i < collection.size(); i++) {} // not this
```

The `java.util.concurrent` package provides two copy-on-write-based collections: `CopyOnWriteArrayList` and `CopyOnWriteArraySet`. Use the copy-on-write collections when your data sets remain relatively small and the number of read operations and traversals greatly outnumber modifications to the collections. Modifications to the collections (not the elements within) are expensive because the entire internal array must be duplicated for each modification.

A thread-safe collection does not make the elements stored within the collection thread-safe. Just because a collection that contains elements is threadsafe does not mean the elements themselves can be safely modified by multiple threads. You might have to use atomic variables, locks, synchronized code blocks, or immutable (read-only) objects to make the objects referenced by a collection thread-safe.

Concurrent Collections

The `java.util.concurrent` package also contains several concurrent collections that can be concurrently read and modified by multiple threads, but without the copy-on-write behavior seen in the copy-on-write collections. The concurrent collections include

- `ConcurrentHashMap`
- `ConcurrentLinkedDeque`
- `ConcurrentLinkedQueue`
- `ConcurrentSkipListMap`

- `ConcurrentSkipListSet`

Be aware that an `Iterator` for a concurrent collection is weakly consistent; it can return elements from the point in time the `Iterator` was created or later. This means that while you are looping through a concurrent collection, you might observe elements that are being inserted by other threads. In addition, you may observe only some of the elements that another thread is inserting with methods such as `addAll` when concurrently reading from the collection. Similarly, the `size` method may produce inaccurate results. Imagine attempting to count the number of people in a checkout line at a grocery store. While you are counting the people in line, some people may join the line and others may leave. Your count might end up close but not exact by the time you reach the end. This is the type of behavior you might see with a weakly consistent collection. The benefit to this type of behavior is that it is permissible for multiple threads to concurrently read and write a collection without having to create multiple internal copies of the collection, as is the case in a copy-on-write collection. If your application cannot deal with these inconsistencies, you might have to use a copy-on-write collection.

The `ConcurrentHashMap` and `ConcurrentSkipListMap` classes implement the `ConcurrentMap` interface. A `ConcurrentMap` enhances a `Map` by adding the atomic `putIfAbsent`, `remove`, and `replace` methods. For example, the `putIfAbsent` method is equivalent to performing the following code as an atomic operation:

```
if (!map.containsKey(key))
    return map.put(key, value);
  else
    return map.get(key);
```

`ConcurrentSkipListMap` and `ConcurrentSkipListSet` are sorted. `ConcurrentSkipListMap` keys and `ConcurrentSkipListSet` elements require the use of the `Comparable` or `Comparator` interfaces to enable ordering.

Blocking Queues

The copy-on-write and the concurrent collections are centered on the idea of multiple threads sharing data. Sometimes, instead of shared data (objects), you need to transfer data between two threads. A `BlockingQueue` is a type of shared collection that is used to exchange data between two or more threads while causing one or more of the threads to wait until the point in time when the data can be exchanged. One use case of a `BlockingQueue` is called the producer-consumer problem. In a producer-consumer scenario, one thread produces data, then adds it to a queue, and another thread must consume the data from the queue. A queue provides the means for the producer and the consumer to exchange objects. The `java.util.concurrent` package provides several

`BlockingQueue` implementations. They include

- `ArrayBlockingQueue`
- `LinkedBlockingDeque`
- `LinkedBlockingQueue`
- `PriorityBlockingQueue`
- `DelayQueue`
- `LinkedTransferQueue`
- `SynchronousQueue`

General Behavior

A blocking collection, depending on the method being called, may cause a thread to block until another thread calls a corresponding method on the collection. For example, if you attempt to remove an element by calling `take()` on any `BlockingQueue` that is empty, the operation will block until another thread inserts an element. Don't call a blocking operation in a thread unless it is safe for that thread to block. The commonly used methods in a `BlockingQueue` are described in the following table.

Method	General Purpose	Unique Behavior
`add(E e)`	Insert an object.	Returns `true` if object added, false if duplicate objects are not allowed. Throws an `IllegalStateException` if the queue is bounded and full.
`offer(E e)`	Insert an object.	Returns `true` if object added, false if the queue is bounded and full.
`put(E e)`	Insert an object.	Returns `void`. If needed, will block until space in the queue becomes available.
`offer(E e, long timeout, TimeUnit unit)`	Insert an object.	Returns `false` if the object was not able to be inserted before the time indicated by the second and third parameters.
`remove(Object o)`	Remove an object.	Returns `true` if an equal object was found in the queue and removed; otherwise, returns `false`.

Method	General Purpose	Unique Behavior
`poll(long timeout, TimeUnit unit)`	Remove an object.	Removes the first object in the queue (the head) and returns it. If the timeout expires before an object can be removed because the queue is empty, a null will be returned.
`take()`	Remove an object.	Removes the first object in the queue (the head) and returns it, blocking if needed until an object becomes available.
`poll()`	Remove an object.	Removes the first object in the queue (the head) and returns it or returns null if the queue is empty.
`element()`	Retrieve an object.	Gets the head of the queue without removing it. Throws a `NoSuchElementException` if the queue is empty.
`peek()`	Retrieve an object.	Gets the head of the queue without removing it. Returns a null if the queue is empty.

Bounded Queues

`ArrayBlockingQueue`, `LinkedBlockingDeque`, and `LinkedBlockingQueue` support a bounded capacity and will block on `put(e)` and similar operations if the collection is full. `LinkedBlockingQueue` is optionally bounded, depending on the constructor you use.

```
BlockingQueue<Integer> bq = new ArrayBlockingQueue<>(1);
try {
  bq.put(42);
  bq.put(43); // blocks until previous value is removed
} catch (InterruptedException ex) {
  // log and handle
}
```

Special-Purpose Queues

A `SynchronousQueue` is a special type of bounded blocking queue; it has a capacity of zero. Having a zero capacity, the first thread to attempt either an insert or remove operation on a `SynchronousQueue` will block until another thread performs the opposite operation. You use a `SynchronousQueue` when you need threads to meet up and exchange an object.

A `DelayQueue` is useful when you have objects that should not be consumed until a specific time. The elements added to a `DelayQueue` will implement the `java.util.concurrent.Delayed` interface, which defines a single method: `public long getDelay(TimeUnit unit)`. The elements of a `DelayQueue` can only be taken once their delay has expired.

The LinkedTransferQueue

A `LinkedTransferQueue` (added in Java 7) is a superset of `ConcurrentLinkedQueue`, `SynchronousQueue`, and `LinkedBlockingQueue`. It can function as a concurrent `Queue` implementation similar to `ConcurrentLinkedQueue`. It also supports unbounded blocking (consumption blocking) similar to `LinkedBlockingQueue` via the `take()` method. Like a `SynchronousQueue`, a `LinkedTransferQueue` can be used to make two threads rendezvous to exchange an object. Unlike a `SynchronousQueue`, a `LinkedTransferQueue` has internal capacity, so the `transfer(E)` method is used to block until the inserted object (and any previously inserted objects) is consumed by another thread.

In other words, a `LinkedTransferQueue` might do almost everything you need from a `Queue`.

Because a `LinkedTransferQueue` implements the `BlockingQueue`, `TransferQueue`, and `Queue` interfaces, it can be used to showcase all the different methods that can be used to add and remove elements using the various types of queues. Creating a `LinkedTransferQueue` is easy. Because `LinkedTransferQueue` is not bound by size, a limit to the number of elements CANNOT be supplied to its constructor.

```
TransferQueue<Integer> tq =
  new LinkedTransferQueue<>(); // not bounded
```

There are many methods to add a single element to a `LinkedTransferQueue`. Note that any method that blocks or waits for any period may throw an `InterruptedException`.

```
boolean b1 = tq.add(1);              // returns true if added or throws
                                     // IllegalStateException if full
tq.put(2);                        // blocks if bounded and full
boolean b3 = tq.offer(3);            // returns true if added or false
                                     // if bounded and full
                                     // recommended over add
boolean b4 =
  tq.offer(4, 10, MILLISECONDS);     // returns true if added
                                     // within the given time
                                     // false if bound and full
tq.transfer(5);                   // blocks until this element is consumed
boolean b6 = tq.tryTransfer(6);        // returns true if consumed
                                       // by an awaiting thread or
                                       // returns false without
                                       // adding if there was no
                                       // awaiting consumer
boolean b7 =
  tq.tryTransfer(7, 10, MILLISECONDS); // will wait the
                                       // given time for
                                       // a consumer
```

Shown next are the various methods to access a single value in a `LinkedTransferQueue`. Again, any method that blocks or waits for any period may

throw an `InterruptedException`.

```
Integer i1 = tq.element();   // gets without removing
                             // throws NoSuchElementException
                             // if empty
Integer i2 = tq.peek();      // gets without removing
                             // returns null if empty
Integer i3 = tq.poll();      // removes the head of the queue
                             // returns null if empty
Integer i4 =
  tq.poll(10, MILLISECONDS); // removes the head of the
                             // queue, waits up to the time
                             // specified before returning
                             // null if empty
Integer i5 = tq.remove();    // removes the head of the queue
                             // throws NoSuchElementException
                             // if empty
Integer i6 = tq.take();      // removes the head of the queue
                             // blocks until an element is ready
```

Use a ***`LinkedTransferQueue`*** *(added in Java 7) instead of another comparable queue type. The other* ***`java.util.concurrent`*** *queues (introduced in Java 5) are less efficient than* ***`LinkedTransferQueue`****.*

Controlling Threads with CyclicBarrier

Whereas blocking queues force threads to wait based on capacity or until a method is called on the queue, a `CyclicBarrier` can force threads to wait at a specific point in the execution until all threads reach that point before continuing. You can think of that point

in the execution as a *barrier:* no thread can proceed beyond that point until all other threads have also reached it.

Imagine you have two threads that are processing data in arrays and you want to copy the processed data from the arrays to a final `ArrayList` that contains data from both threads. You can't have both threads accessing the final `ArrayList` because `ArrayList` is not thread-safe. It is better to wait until both threads are done processing the arrays and only then copy the data to the final `ArrayList` using one thread.

This is an example of the type of problem where `CyclicBarrier` can be useful. When you can break up a problem into smaller pieces that can be processed concurrently and then combine the data into a result at certain key points in the processing, using a `CyclicBarrier` ensures the threads wait for each other at those key points in the execution.

Let's take a look at some code to see how to use `CyclicBarrier` to force two threads to wait for each other. In this example, we'll use the optional `Runnable` that `CyclicBarrier` will take and run, once both threads reach the barrier point. This `Runnable` is run only one time, by the last thread to reach the barrier and before any of the threads can continue, ensuring the code in the `Runnable` is thread-safe. Each thread processes a small array and then ends, but imagine how this idea might be applied to threads processing large arrays a chunk at a time and continuing after a barrier is reached. In fact, that's exactly why `CyclicBarrier` is named "cyclic": the `CyclicBarrier` can be reused after the threads are released to continue running.

```
public class CB {
   List<String> result = new ArrayList<>();
   static String[] dogs1 = {"boi", "clover", "charis"};
   static String[] dogs2 = {"aiko", "zooey", "biscuit"};
   final CyclicBarrier barrier;           // The barrier for the threads
   class ProcessDogs implements Runnable { // Each thread will process
      String dogs[];                      // an array of dogs
      ProcessDogs(String[] d) { dogs = d; }
// #4
      public void run() {                 // Convert first chars into
                                          // uppercase
         for (int i = 0; i < dogs.length; i++) {
            String dogName = dogs[i];
            String newDogName = dogName.substring(0, 1).toUpperCase()
                              + dogName.substring(1);
            dogs[i] = newDogName;
         }
         try {
// #5
            barrier.await();              // Wait at the barrier
         } catch(InterruptedException | BrokenBarrierException e) {
            // The other thread must have been interrupted
            e.printStackTrace();
         }
```

```
// #7
        System.out.println(Thread.currentThread().getName() + " is done!");
      }
   }
   public CB() {
// #1   ---------------------------- the 2nd argument code runs later
      barrier = new CyclicBarrier(2, () -> { // 2 threads, 1 Runnable
// #6
                                            // Copy results to list
         for (int i = 0; i < dogs1.length; i++) {
            result.add(dogs1[i]);           // add dogs from array 1
         }
         for (int i = 0; i < dogs2.length; i++) {
            result.add(dogs2[i]);           // add dogs from array 2
         }
                                            // print the thread name and
                                            // result
         System.out.println(Thread.currentThread().getName() +
                            " Result: " + result);
      });
// #2
      Thread t1 = new Thread(new ProcessDogs(dogs1));
      Thread t2 = new Thread(new ProcessDogs(dogs2));
// #3
      t1.start();
      t2.start();
      System.out.println("Main Thread is done");
   }
   public static void main(String[] args) {
      CB cb = new CB();
   }
}
```

In this example, we have two arrays of dog names and the processing is quite simple; we just convert the first letter of each dog name from lowercase to uppercase. Once both arrays have been processed, we then combine the results into an `ArrayList` of dog names. In the class `CB`, we define the `result ArrayList`, where we'll write results when both threads have finished processing the two arrays, `dogs1` and `dogs2`. We also declare the `CyclicBarrier` that both threads will use to wait for each other. Let's step through how the code executes (follow the numbers in the code):

1. In the `CB` constructor, we create a new `CyclicBarrier`, passing in the number of threads that will wait at the `barrier` and a Runnable that will be run by the last thread to reach the barrier. Here, we're using a lambda expression for this `Runnable`.
2. We then create two threads, `t1` and `t2`, and pass each the `ProcessDogs Runnable`. When we create the `ProcessDogs Runnable`s for the threads, we pass in the array that the thread will process: we pass `dogs1` to thread `t1` and `dogs2` to thread `t2`.
3. We then start both threads and print a message saying the main thread is done.
4. The two threads begin running and process their respective arrays. The `ProcessDogs run()` method simply iterates through the array and converts the first letter of the dog name to uppercase, storing the modified string back in the original array.
5. Once the loop is complete, the thread then waits at the barrier by calling the barrier's `await()` method. We'll come back and talk about the exception handling in a minute.
6. Once both threads reach the barrier, then the `Runnable` we passed to the `CyclicBarrier` constructor is executed by the last thread to reach the barrier. This adds all the items from both arrays to the `ArrayList result` and prints the `result`.
7. When the `Runnable` completes, then both threads are released and can continue executing where they left off at the barrier. In this example, each thread just prints a message to the console indicating it's done (with the thread name), but you can imagine that in a more realistic example, they could continue on to do more processing. Notice that neither thread can print out that it's done until *both* threads have reached the barrier and the barrier `Runnable` is complete.

If you have a machine with at least two cores and you run this code several times, you may see thread 0 displaying the result or you may see thread 1 displaying the result. It just depends on which thread gets to the barrier last. Also, notice that the main thread often finishes before the other threads, but not always. The most important point to notice is that

every time you run this code, you will see the resulting `ArrayList` displayed *before* the messages from the threads that they are done. Why? Because both threads must wait at the barrier for the barrier `Runnable` to complete before they can continue!

```
Main Thread is done
Thread-0 Result: [Boi, Clover, Charis, Aiko, Zooey, Biscuit]
Thread-0 is done!
Thread-1 is done!
```

Now let's get back to the exception handling on the `barrier.await()` method. What if one of the threads gets stuck? If this happens, then all the other threads waiting at the barrier get an `InterruptedException` or a `BrokenBarrierException` and are all released. In that case, the barrier `Runnable` will not run and the remaining threads will continue.

You can reuse a `CyclicBarrier` or use multiple `CyclicBarriers` to coordinate threads at more than one barrier point. For example, you might have the thread `Runnable` execute some code, wait at barrier 1, then execute some more code, wait at barrier 2, and then finally execute more code and finish. To reset a barrier to its initial state, call the `reset()` method. If you need the main thread to also wait at the barrier, you can call `await()` on the barrier in the main thread, but don't forget to increase the number of threads allowed at the barrier by one when you create the `CyclicBarrier`.

CERTIFICATION OBJECTIVE

Use Executors and ThreadPools (OCP Objective 10.1)

10.1 Create worker threads using Runnable, Callable and use an ExecutorService to concurrently execute tasks.

`Executors` (and the `ThreadPools` used by them) help meet two of the same needs as `Threads` do:

1. Creating and scheduling some Java code for execution and
2. Optimizing the execution of that code for the hardware resources you have available (using all CPUs, for example)

With traditional threading, you handle needs 1 and 2 yourself. With `Executors`, you handle need 1, but you get to use an off-the-shelf solution for need 2. The `java.util.concurrent` package provides several different off-the-shelf solutions (`Executors` and `ThreadPools`), which you'll read about in this chapter.

When you have multiple needs or concerns, it is common to separate the code for each need into different classes. This makes your application more modular and flexible. This is a fundamental programming principle called "separation of concerns."

In a way, an `Executor` is an alternative to starting new threads. Using `Threads` directly can be considered low-level multithreading, whereas using `Executors` can be considered high-level multithreading. To understand how an `Executor` can replace manual thread creation, let us first analyze what happens when starting a new thread.

1. First, you must identify a task of some sort that forms a self-contained unit of work. You will typically code this task as a class that implements the `Runnable` interface.
2. After creating a `Runnable`, the next step is to execute it. You have two options for executing a `Runnable`:
   - Option one Call the `run` method synchronously (i.e., without starting a thread). This is probably not what you would normally do.

     ```
     Runnable r = new MyRunnableTask();
     r.run(); // executed by calling thread
     ```

   - Option two Call the method indirectly, most likely with a new thread.

     ```
     Runnable r = new MyRunnableTask();
     Thread t1 = new Thread(r);
     t1.start();
     ```

The second approach has the benefit of executing your task asynchronously, meaning the primary flow of execution in your program can continue executing, without waiting for the task to complete. On a multiprocessor system, you must divide a program into a collection of asynchronous tasks that can execute 747concurrently in order to take advantage of all of the computing power a system possesses.

Identifying Parallel Tasks

Some applications are easier to divide into separate tasks than others. A single-user desktop application may only have a handful of tasks that are suitable for concurrent execution. Networked multiuser servers, on the other hand, have a natural division of work. Each user's actions can be a task. Continuing our computer game scenario, imagine a computer program that can play chess against thousands of people simultaneously. Each player

submits his or her move, the computer calculates its move, and finally it informs the player of that move.

Why do we need an alternative to `new Thread(r).start()`? What are the drawbacks? If we use our online chess game scenario, then having 10,000 concurrent players might mean 10,001 concurrent threads. (One thread awaits network connections from clients and performs a `Thread(r).start()` for each player.) The player thread would be responsible for reading the player's move, computing the computer's move, and making the response.

How Many Threads Can You Run?

Do you own a computer that can concurrently run 10,000 threads or 1,000 or even 100? Probably not—this is a trick question. A quad-core CPU (with four processors per unit) might be able to execute two threads per core for a total of eight concurrently executing threads. You can start 10,000 threads, but not all of them will be running at the same time. The underlying operating system's task scheduler rotates the threads so that they each get a slice of time on a processor. Ten thousand threads all competing for a turn on a processor wouldn't make for a very responsive system. Threads would either have to wait so long for a turn or get such small turns (or both) that performance would suffer.

In addition, each thread consumes system resources. It takes processor cycles to perform a context switch (saving the state of a thread and resuming another thread), and each thread consumes system memory for its stack space. Stack space is used for temporary storage and to keep track of where a thread returns to after completing a method call. Depending on a thread's behavior, it might be possible to lower the cost (in RAM) of creating a thread by reducing a thread's stack size.

To reduce a thread's stack size, the Oracle JVM supports using the nonstandard **`-Xss1024k`** *option to the* **`java`** *command. Note that decreasing the value too far can result in some threads throwing exceptions when performing certain tasks, such as making a large number of recursive method calls.*

Another limiting factor in being able to run 10,000 threads in an application has to do with the underlying limits of the OS. Operating systems typically have limits on the number of threads an application can create. These limits can prevent a buggy application from spawning countless threads and making your system unresponsive. If you have a legitimate need to run 10,000 threads, you will probably have to consult your operating system's documentation to discover possible limits and configuration options.

CPU-Intensive vs. I/O-Intensive Tasks

If you correctly configure your OS and you have enough memory for each thread's stack space plus your application's primary memory (heap), will you be able to run an application with 10,000 threads? It depends…. Remember that your processor can only run a small number of concurrent threads (in the neighborhood of 8 to 16 threads). Yet many network server applications, such as our online chess game, would have traditionally started a new thread for each connected client. A system might be able to run an application with such a high number of threads because most of the threads are not doing anything. More precisely, in an application like our online chess server, most threads would be blocked waiting on I/O operations such as `InputStream.read` or `OutputStream.write` method calls.

When a thread makes an I/O request using `InputStream.read` and the data to be read isn't already in memory, the calling thread will be put to sleep by the system until the requested data can be loaded. This is much more efficient than keeping the thread on the processor while it has nothing to do. I/O operations are extremely slow when compared to compute operations—reading a sector from a hard drive takes much longer than adding hundreds of numbers. A processor might execute hundreds of thousands, or even millions, of instructions while awaiting the completion of an I/O request. The type of work (either CPU intensive or I/O intensive) a thread will be performing is important when considering how many threads an application can safely run. Imagine your world-class-computer 749chess-playing program takes one minute of processor time (no I/O at all) to calculate each move. In this scenario, it would only take about 16 concurrent players to cause your system to have periods of maximum CPU utilization.

If your tasks will be performing I/O operations, you should be concerned about how increased load (users) might affect scalability. If your tasks perform blocking I/O, then you might need to utilize a thread-per-task model. If you don't, then all your threads may be tied up in I/O operations with no threads remaining to support additional users. Another option would be to investigate whether you can use nonblocking I/O instead of blocking I/O.

Fighting for a Turn

If it takes the computer player one minute to calculate a turn and it takes a human player about the same time, then each player only uses one minute of CPU time out of every two minutes of real time. With a system capable of executing 16 concurrent game threads, that means we could handle 32 connected players. But if all 32 players make their turn at once, the computer will be stuck trying to calculate 32 moves at once. If the system uses

preemptive multitasking (the most common type), then each thread will get preempted while it is running (paused and kicked off the CPU) so a different thread can take a turn (time slice). In most JVM implementations, this is handled by the underlying operating system's task scheduler. The task scheduler is itself a software program. The more CPU cycles spent scheduling and preempting threads, the less processor time you have to execute your application threads. Note that it would appear to the untrained observer that all 32 threads were running concurrently because a preemptive multitasking system will switch out the running threads frequently (millisecond time slices).

Decoupling Tasks from Threads

The best design would be one that utilized as many system resources as possible without attempting to overutilize the system. If 16 threads are all you need to fully utilize your CPU, why would you start more than that? In a traditional system, you start more threads than your system can concurrently run and hope that only a small number are in a running state. If we want to adjust the number of threads that are started, we need to decouple the tasks that are to be performed (our `Runnable` instances) from our thread creation and starting. This is where a `java.util.concurrent.Executor` can help. The basic usage looks something like this:

```
Runnable r = new MyRunnableTask();
Executor ex = // details to follow
ex.execute(r);
```

A `java.util.concurrent.Executor` is used to execute the `run` method in a `Runnable` instance much like a thread. Unlike a more traditional `new Thread(r).start()`, an `Executor` can be designed to use any number of threading approaches, including

- Not starting any threads at all (task is run in the calling thread)
- Starting a new thread for each task
- Queuing tasks and processing them with only enough threads to keep the CPU utilized

You can easily create your own implementations of an `Executor` with custom behaviors. As you'll see soon, several implementations are provided in the standard Java SE libraries. Looking at sample `Executor` implementations can help you to understand their behavior. This next example doesn't start any new threads; instead, it executes the `Runnable` using the thread that invoked the `Executor`.

```
import java.util.concurrent.Executor;
public class SameThreadExecutor implements Executor {
    @Override
    public void execute(Runnable command) {
        command.run(); // caller waits
    }
}
```

The following `Executor` implementation would use a new thread for each task:

```
import java.util.concurrent.Executor;
public class NewThreadExecutor implements Executor {
    @Override
    public void execute(Runnable command) {
        Thread t = new Thread(command);
        t.start();
    }
}
```

This example shows how an `Executor` implementation can be put to use:

```
Runnable r = new MyRunnableTask();
Executor ex = new NewThreadExecutor(); // choose Executor
ex.executor(r);
```

By coding to the `Executor` interface, the submission of tasks is decoupled from the execution of tasks. The result is that you can easily modify how threads are used to execute tasks in your applications.

There is no "right number" of threads for task execution. The type of task (CPU intensive

versus I/O intensive), number of tasks, I/O latency, and system resources all factor into determining the ideal number of threads to use. You should test your applications to determine the ideal threading model. This is one reason why the ability to separate task submission from task execution is important.

Several `Executor` implementations are supplied as part of the standard Java libraries. The `Executors` class (notice the "s" at the end) is a factory for `Executor` implementations.

```
Runnable r = new MyRunnableTask();
Executor ex = Executors.newCachedThreadPool(); // choose Executor
ex.execute(r);
```

The `Executor` instances returned by `Executors` are actually of type `ExecutorService` (which extends `Executor`). An `ExecutorService` provides management capability and can return `Future` instances that are used to obtain the result of executing a task asynchronously. We'll talk more about `Future` in a few pages!

```
Runnable r = new MyRunnableTask();
ExecutorService ex = Executors.newCachedThreadPool(); // subtype of Executor
ex.execute(r);
```

Three types of `ExecutorService` instances can be created by the factory methods in the `Executors` class: cached thread pool executors, fixed thread pool executors, and single thread pool executors.

Cached Thread Pools

```
ExecutorService ex = Executors.newCachedThreadPool();
```

A cached thread pool will create new threads as they are needed and reuse threads that have become free. Threads that have been idle for 60 seconds are removed from the pool.

Watch out! Without some type of external limitation, a cached thread pool may be used to create more threads than your system can handle.

Fixed Thread Pools—Most Common

```
ExecutorService ex = Executors.newFixedThreadPool(4);
```

A fixed thread pool is constructed using a numeric argument (4 in the preceding example) that specifies the number of threads used to execute tasks. This type of pool will probably be the one you use the most because it prevents an application from overloading a

system with too many threads. Tasks that cannot be executed immediately are placed on an unbounded queue for later execution.

You might base the number of threads in a fixed thread pool on some attribute of the system your application is executing on. By tying the number of threads to system resources, you can create an application that scales with changes in system hardware. To query the number of available processors, you can use the ***`java.lang.Runtime`*** *class.*

```
Runtime rt = Runtime.getRuntime();
int cpus = rt.availableProcessors();
```

ThreadPoolExecutor

Both `Executors.newCachedThreadPool()` and `Executors.newFixedThreadPool(4)` return objects of type `java.util.concurrent.ThreadPoolExecutor` (which implements `ExecutorService` and `Executor`). You will typically use the `Executors` factory methods instead of creating `ThreadPoolExecutor` instances directly, but you can cast the fixed or cached thread pool `ExecutorService` references if you need access to the additional methods. The following example shows how you could dynamically adjust the thread count of a pool at runtime:

```
ThreadPoolExecutor tpe = (ThreadPoolExecutor)Executors.newFixedThreadPool(4);
tpe.setCorePoolSize(8);
tpe.setMaximumPoolSize(8);
```

Single Thread Pools

```
ExecutorService ex = Executors.newSingleThreadExecutor();
```

A single thread pool uses a single thread to execute tasks. Tasks that cannot be executed immediately are placed on an unbounded queue for later execution. Unlike a fixed thread pool executor with a size of 1, a single thread executor prevents any adjustments to the number of threads in the pool.

Scheduled Thread Pool

In addition to the three basic `ExecutorService` behaviors outlined already, the Executors class has factory methods to produce a `ScheduledThreadPoolExecutor`. A

`ScheduledThreadPoolExecutor` enables tasks to be executed after a delay or at repeating intervals. Here, we see some thread-scheduling code in action:

```
ScheduledExecutorService ftses =
  Executors.newScheduledThreadPool(4);                    // multi-threaded
                                                          // version
ftses.schedule(r, 5, TimeUnit.SECONDS);                   // run once after
                                                          // a delay
ftses.scheduleAtFixedRate(r, 2, 5, TimeUnit.SECONDS);     // begin after a
                                                          // 2sec delay
  // and begin again every 5 seconds
ftses.scheduleWithFixedDelay(r, 2, 5, TimeUnit.SECONDS); // begin after
                                                          // 2sec delay
  // and begin again 5 seconds *after* completing the last execution
```

The Callable Interface

So far, the `Executors` examples have used a `Runnable` instance to represent the task to be executed. The `java.util.concurrent.Callable` interface serves the same purpose as the `Runnable` interface, but provides more flexibility. Unlike the `Runnable` interface, a `Callable` may return a result upon completing execution and may throw a checked exception. An `ExecutorService` can be passed a `Callable` instead of a `Runnable`.

Avoid using methods such as ***`Object.wait`, `Object.notify`,*** *and* ***`Object.notifyAll`*** *in tasks (****`Runnable`*** *and* ***`Callable`*** *instances) that are submitted to an* ***`Executor`*** *or* ***`ExecutorService`****. Because you might not know what the threading behavior of an* ***`Executor`*** *is, it is a good idea to avoid operations that may interfere with thread execution. Avoiding these types of methods is advisable anyway since they are easy to misuse.*

The primary benefit of using a Callable is the ability to return a result. Because an `ExecutorService` may execute the `Callable` asynchronously (just like a `Runnable`), you need a way to check the completion status of a `Callable` and obtain the result later. A

`java.util.concurrent.Future` is used to obtain the status and result of a `Callable`. Without a `Future`, you'd have no way to obtain the result of a completed `Callable` and you might as well use a `Runnable` (which returns void) instead of a `Callable`. Here's a simple `Callable` example that loops a random number of times and returns the random loop count:

```
import java.util.concurrent.Callable;
import java.util.concurrent.ThreadLocalRandom;
public class MyCallable implements Callable<Integer> {

  @Override
  public Integer call() {
    // Obtain a random number from 1 to 10
    int count = ThreadLocalRandom.current().nextInt(1, 11);
    for(int i = 1; i <= count; i++) {
      System.out.println("Running..." + i);
    }
    return count;
  }
}
```

Submitting a `Callable` to an `ExecutorService` returns a `Future` reference. When you use the `Future` to obtain the `Callable`'s result, you will have to handle two possible exceptions:

- InterruptedException Raised when the thread calling the `Future`'s `get()` method is interrupted before a result can be returned
- ExecutionException Raised when an exception was thrown during the execution of the `Callable`'s `call()` method

```
Callable<Integer> c = new MyCallable();
ExecutorService ex =
        Executors.newCachedThreadPool();
Future<Integer> f = ex.submit(c); // finishes in the future
try {
  Integer v = f.get(); // blocks until done
  System.out.println("Ran:" + v);
} catch (InterruptedException | ExecutionException iex) {
  System.out.println("Failed");
}
```

I/O activities in your ***Runnable*** *and* ***Callable*** *instances can be a serious bottleneck. In preceding examples, the use of* ***System.out.println()*** *will cause I/O activity. If this wasn't a trivial example being used to demonstrate* ***Callable*** *and* ***ExecutorService****, you would probably want to avoid repeated calls to* ***println()*** *in the* ***Callable****. One possibility would be to use* ***StringBuilder*** *to concatenate all output strings and have a single* ***println()*** *call before the* ***call()*** *method returns. Another possibility would be to use a logging framework (see* ***java.util.logging****) in place of any* ***println()*** *calls.*

ThreadLocalRandom

The first `Callable` example used a `java.util.concurrent.ThreadLocalRandom`. `ThreadLocalRandom` was introduced in Java 7 as a new way to create random numbers. `Math.random()` and shared `Random` instances are thread-safe, but suffer from contention when used by multiple threads. A `ThreadLocalRandom` is unique to a thread and will perform better because it avoids any contention. `ThreadLocalRandom` also provides several convenient methods such as `nextInt(int,int)` that allow you to specify the range of possible values returned.

ExecutorService Shutdown

You've seen how to create `Executors` and how to submit `Runnable` and `Callable` tasks to those `Executors`. The final component to using an `Executor` is shutting it down once it is done processing tasks. An `ExecutorService` should be shut down once it is no

longer needed to free up system resources and to allow graceful application shutdown. Because the threads in an `ExecutorService` may be nondaemon threads, they may prevent normal application termination. In other words, your application stays running after completing its main method. You could perform a `System.exit(0)` call, but it would preferable to allow your threads to complete their current activities (especially if they are writing data).

```
ExecutorService ex =
// …
ex.shutdown(); // no more new tasks
               // but finish existing tasks
try {
  boolean term = ex.awaitTermination(2, TimeUnit.SECONDS);
    // wait 2 seconds for running tasks to finish
} catch (InterruptedException ex1) {
    // did not wait the full 2 seconds
} finally {
  if(!ex.isTerminated()) // are all tasks done?
  {
    List<Runnable> unfinished = ex.shutdownNow();
    // a collection of the unfinished tasks
  }
}
```

For long-running tasks (especially those with looping constructs), consider using `Thread.currentThread().isInterrupted()` to determine if a `Runnable` or `Callable` should return early. The `ExecutorService.shutdownNow()` method will typically call `Thread.interrupt()` in an attempt to terminate any unfinished tasks.

CERTIFICATION OBJECTIVE

Use the Parallel Fork/Join Framework (OCP Objective 10.5)

The Fork-Join Framework provides a highly specialized `ExecutorService`. The other `ExecutorService` instances you've seen so far are centered on the concept of submitting multiple tasks to an `ExecutorService`. By doing this, you provide an easy avenue for an `ExecutorService` to take advantage of all the CPUs in a system by using threads to complete tasks. Sometimes, you don't have multiple tasks; instead, you have one really big task.

There are many large tasks or problems you might need to solve in your application. For example, you might need to initialize the elements of a large array with values. You might think that initializing an array doesn't sound like a large complex task in need of a framework. The key is that it needs to be a large task. What if you need to fill up a 100,000,000-element array with randomly generated values? The Fork/Join Framework makes it easier to tackle big tasks like this, while leveraging all of the CPUs in a system.

Divide and Conquer

Certain types of large tasks can be split up into smaller subtasks; those subtasks might, in turn, be split up into even smaller tasks. There is no limit to how many times you might subdivide a task. For example, imagine the task of having to repaint a single long fence that borders several houses. The "paint the fence" task could be subdivided so that each household would be responsible for painting a section of the fence. Each household could then subdivide their section into subsections to be painted by individual family members. In this example, there are three levels of recursive calls. The calls are considered recursive because at each step we are trying to accomplish the same thing: paint the fence. In other words, Joe, one of the home owners, was told by his wife, "paint that (huge) fence; it looks old." Joe decides that painting the whole fence is too much work and talks all the households along the fence into taking a subsection. Now Joe is telling himself "paint that (subsection of) fence; it looks old." Again, Joe decides that it is still too much work and subdivides his section into smaller sections for each member of his household. Again, Joe tells himself "paint that (subsection of) fence; it looks old," but this time, he decides that the amount of work is manageable and proceeds to paint his section of fence. Assuming everyone else paints their subsections (hopefully in a timely fashion), the result is the entire fence being painted.

When using the Fork/Join Framework, your tasks will be coded to decide how many levels of recursion (how many times to subdivide) are appropriate. You'll want to split things up into enough subtasks that you have adequate tasks to keep all of your CPUs utilized. Sometimes,

the best number of tasks can be a little hard to determine because of factors we will discuss later. You might have to benchmark different numbers of task divisions to find the optimal number of subtasks that should be created.

Just because you can use Fork/Join to solve a problem doesn't always mean you should. If our initial task is to paint eight fence planks, then Joe might just decide to paint them himself. The effort involved in subdividing the problem and assigning those tasks to workers (threads) can sometimes be more than the actual work you want to perform. The number of elements (or fence planks) is not the only thing to consider—the amount of work performed on each element is also important. Imagine if Joe was asked to paint a mural on each fence plank. Because processing each element (fence plank) is so time consuming, in this case, it might be beneficial to adopt a divide-and-conquer solution even though there is a small number of elements.

ForkJoinPool

The Fork/Join `ExecutorService` implementation is `java.util.concurrent.ForkJoinPool`. You will typically submit a single task to a `ForkJoinPool` and await its completion. The `ForkJoinPool` and the task itself work together to divide and conquer the problem. Any problem that can be recursively divided can be solved using Fork/Join. Anytime you want to perform the same operation on a collection of elements (painting thousands of fence planks or initializing 100,000,000 array elements), consider using Fork/Join.

To create a `ForkJoinPool`, simply call its no-arg constructor:

```
ForkJoinPool fjPool = new ForkJoinPool();
```

The no-arg `ForkJoinPool` constructor creates an instance that will use the `Runtime.availableProcessors()` method to determine the level of parallelism. The level of parallelism determines the number of threads that will be used by the `ForkJoinPool`.

There is also a `ForkJoinPool(int parallelism)` constructor that allows you to override the number of threads that will be used.

ForkJoinTask

Just as with `Executors`, you must capture the task to be performed as Java code. With the Fork/Join Framework, a `java.util.concurrent.ForkJoinTask` instance (actually a subclass—more on that later) is created to represent the task that should be accomplished. This is different from other executor services that primarily used either `Runnable` or `Callable`. A `ForkJoinTask` concrete subclass has many methods (most of which you will never use), but the following methods are important: compute(), fork(), and join().

A `ForkJoinTask` subclass is where you will perform most of the work involved in

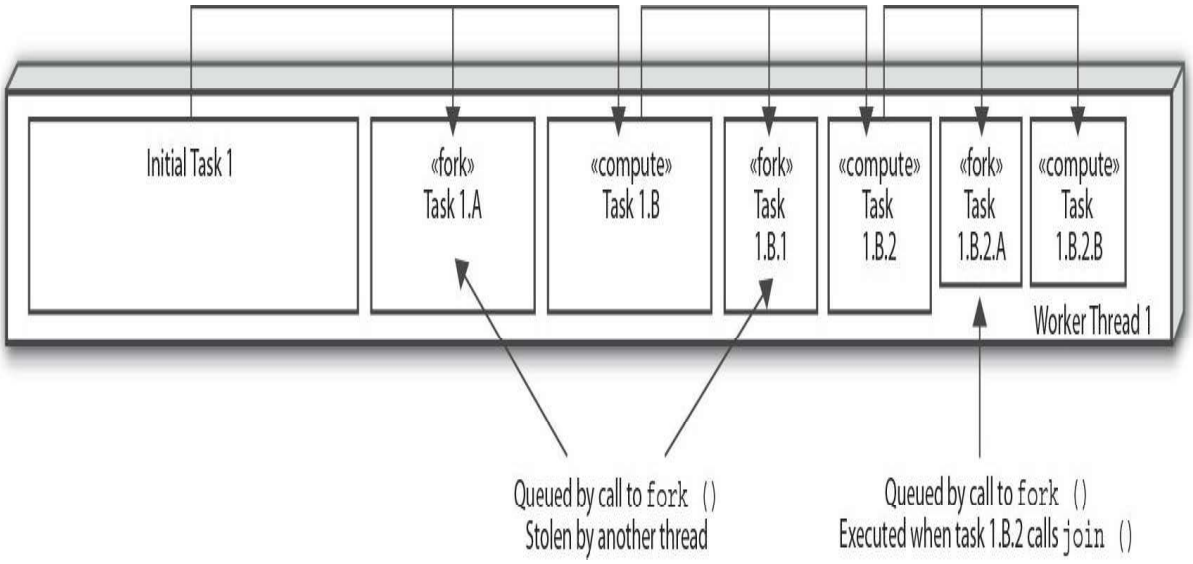
completing a Fork/Join task. `ForkJoinTask` is an abstract base class; we will discuss the two subclasses, `RecursiveTask` and `RecursiveAction`, later. The basic structure of any `ForkJoinTask` is shown in this pseudocode example:

```
class ForkJoinPaintTask {
  compute() {
    if(isFenceSectionSmall()) {  // is it a manageable amount of work?
      paintFenceSection();       // do the task
    } else {                     // task too big, split it
      ForkJoinPaintTask leftHalf = getLeftHalfOfFence();
      leftHalf.fork();           // queue left half of task
      ForkJoinPaintTask rightHalf = getRightHalfOfFence();
      rightHalf.compute();       // work on right half of task
      leftHalf.join();           // wait for queued task to be complete
    }
  }
}
```

Fork

With the Fork/Join Framework, each thread in the **`ForkJoinPool`** has a queue of the tasks it is working on; this is unlike most **`ExecutorService`** implementations that have a single shared task queue. The `fork()` method places a `ForkJoinTask` in the current thread's task queue. A normal thread does not have a queue of tasks—only the specialized threads in a `ForkJoinPool` do. This means that you can only call `fork()` if you are within a `ForkJoinTask` that is being executed by a `ForkJoinPool`.

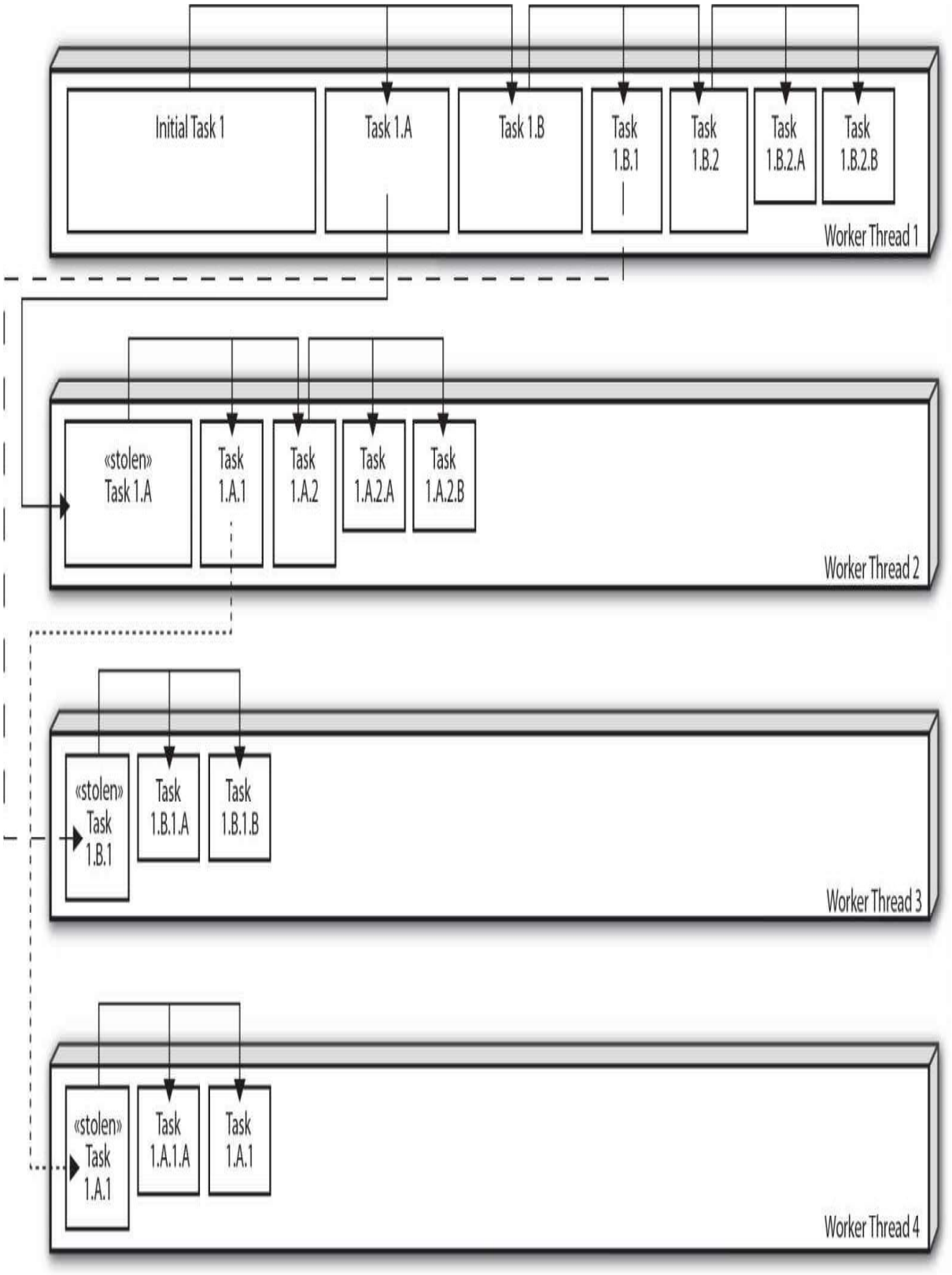
Initially, only a single thread in a `ForkJoinPool` will be busy when you submit a task. That thread will begin to subdivide the tasks into smaller tasks. Each time a task is subdivided into two subtasks, you fork (or queue) the first task and compute the second task. In the event you need to subdivide a task into more than two subtasks, each time you split a task, you would fork every new subtask except one (which would be computed).



Work Stealing

Notice how the call to `fork()` is placed before the call to `compute()` or `join()`. A key feature of the Fork/Join Framework is work stealing. Work stealing is how the other threads in a `ForkJoinPool` will obtain tasks. When initially submitting a Fork/Join task for execution, a single thread from a `ForkJoinPool` begins executing (and subdividing) that task. Each call to `fork()` placed a new task in the calling thread's task queue. The order in which the tasks are queued is important. The tasks that have been queued the longest represent larger amounts of work. In the `ForkJoinPaintTask` example, the task that represents 100 percent of the work would begin executing, and its first queued (forked) task would represent 50 percent of the fence, the next 25 percent, then 12.5 percent, and so on. Of course, this can vary, depending on how many times the task will be subdivided and whether we are 760splitting the task into halves or quarters or some other division, but in this example, we are splitting each task into two parts: queuing one part and executing the second part.

The nonbusy threads in a `ForkJoinPool` will attempt to steal the oldest (and, therefore, largest) task from any Fork/Join thread with queued tasks. Given a `ForkJoinPool` with four threads, one possible sequence of events could be that the initial thread queues tasks that represent 50 percent and 25 percent of the work, which are then stolen by two different threads. The thread that stole the 50 percent task then subdivides that task and places a 25 percent task on its queue, which is then stolen by a fourth thread, resulting in four threads that each process 25 percent of the work.



Of course, if everything was always this evenly distributed, you might not have as much of a need for Fork/Join. You could just presplit the work into a number of tasks equal to the number of threads in your system and use a regular `ExecutorService`. In practice, each of the four threads will not finish their 25 percent of the work at the same time—one thread will be the slow thread that doesn't get as much work done. There are many reasons

for this: The data being processed may affect the amount of computation (25 percent of an array might not mean 25 percent of the workload), or a thread might not get as much time to execute as the other threads. Operating systems and other running applications are also going to consume CPU time. In order to finish executing the Fork/Join task as 761soon as possible, the threads that finish their portions of the work first will start to steal work from the slower threads—this way, you will be able to keep all of the CPU involved. If you only split the tasks into 25 percent of the data (with four threads), then there would be nothing for the faster threads to steal from when they finish early. In the beginning, if the slower thread stole 25 percent of the work and started processing it without further subdividing and queuing, then there would be no work on the slow thread's queue to steal. You should subdivide the tasks into a few more sections than are needed to evenly distribute the work among the number of threads in your `ForkJoinPool`s because threads will most likely not perform exactly the same. Subdividing the tasks is extra work—if you do it too much, you might hurt performance. Subdivide your tasks enough to keep all CPUs busy, but not more than is needed. Unfortunately, there is no magic number to split your tasks into—it varies based on the complexity of the task, the size of the data, and even the performance characteristics of your CPUs.

Back to fence painting, make the `isFenceSectionSmall()` logic as simple as possible (low overhead) and easy to change. You should benchmark your Fork/Join code (using the hardware that you expect the code to typically run on) and find an amount of task subdivision that works well. It doesn't have to be perfect; once you are close to the ideal range, you probably won't see much variation in performance unless other factors come into play (different CPUs, etc.).

Join

When you call `join()` on the (left) task, it should be one of the last steps in the `compute` method, after calling `fork()` and `compute()`. Calling `join()` says, "I can only proceed when this (left) task is done." Several possible things can happen when you call `join()`:

- The task you call `join()` on might already be done. Remember you are calling `join()` on a task that already had `fork()` called. The task might have been stolen and completed by another thread. In this case, calling `join()` just verifies the task is complete and you can continue on.
- The task you call `join()` on might be in the middle of being processed. Another thread could have stolen the task, and you'll have to wait until the joined task is done before continuing.
- The task you call `join()` on might still be in the queue (not stolen). In this case, the thread calling `join()` will execute the joined task.

RecursiveAction

`ForkJoinTask` is an abstract base class that outlines most of the methods, such as `fork()`

and `join()`, in a Fork/Join task. If you need to create a `ForkJoinTask` that does not return a result, then you should subclass `RecursiveAction`. `RecursiveAction` extends `ForkJoinTask` and has a single abstract compute method that you must implement:

```
protected abstract void compute();
```

An example of a task that does not need to return a result would be any task that initializes an existing data structure. The following example will initialize an array to contain random values. Notice that there is only a single array throughout the entire process. When subdividing an array, you should avoid creating new objects when possible.

```
public class RandomInitRecursiveAction extends RecursiveAction {
  private static final int THRESHOLD = 10000;
  private int[] data;
  private int start;
  private int end;

  public RandomInitRecursiveAction(int[] data, int start, int end) {
    this.data = data;
    this.start = start;              // where does our section begin?
    this.end = end;                  // how large is this section?
  }
  @Override
  protected void compute() {
    if (end - start <= THRESHOLD) {  // is it a manageable amount of work?
      // do the task
      for (int i = start; i < end; i++) {
        data[i] =  ThreadLocalRandom.current().nextInt();
      }
    } else {                         // task too big, split it
      int halfWay = ((end - start) / 2) + start;
      RandomInitRecursiveAction a1 =
        new RandomInitRecursiveAction(data, start, halfWay);
      a1.fork(); // queue left half of task
      RandomInitRecursiveAction a2 =
        new RandomInitRecursiveAction(data, halfWay, end);
```

```
            a2.compute();    // work on right half of task
            a1.join();       // wait for queued task to be complete
        }
    }
}
```

Sometimes, you will see one of the `invokeAll` methods from the `ForkJoinTask` class used in place of the fork/compute/join method combination. 763The `invokeAll` methods are convenience methods that can save some typing. Using them will also help you avoid bugs! The first task passed to `invokeAll` will be executed (compute is called), and all additional tasks will be forked and joined. In the preceding example, you could eliminate the three fork/compute/join lines and replace them with a single line:

```
invokeAll(a2, a1);
```

To begin the application, we create a large array and initialize it using Fork/Join:

```
public static void main(String[] args) {
  int[] data = new int[10_000_000];
  ForkJoinPool fjPool = new ForkJoinPool();
  RandomInitRecursiveAction action =
    new RandomInitRecursiveAction(data, 0, data.length);
  fjPool.invoke(action);
}
```

Notice that we do not expect any return values when calling invoke. A `RecursiveAction` returns nothing.

RecursiveTask

If you need to create a `ForkJoinTask` that does return a result, then you should subclass `RecursiveTask`. `RecursiveTask` extends `ForkJoinTask` and has a single abstract compute method that you must implement:

```
protected abstract V compute(); // V is a generic type
```

The following example will find the position in an array with the greatest value; if duplicate values are found, the first occurrence is returned. Notice that there is only a single

array throughout the entire process. (Just like before, when subdividing an array, you should avoid creating new objects when possible.)

```
public class FindMaxPositionRecursiveTask extends RecursiveTask<Integer> {
  private static final int THRESHOLD = 10000;
  private int[] data;
  private int start;
  private int end;
```

```
  public FindMaxPositionRecursiveTask(int[] data, int start, int end) {
    this.data = data;
    this.start = start;
    this.end = end;
  }

  @Override
  protected Integer compute() {     // return type matches the <generic> type
    if (end - start <= THRESHOLD) { // is it a manageable amount of work?
      int position = 0;             // if all values are equal, return
                                    // position 0
      for (int i = start; i < end; i++) {
        if (data[i] > data[position]) {
          position = i;
        }
      }
        return position;
      } else { // task too big, split it
      int halfWay = ((end - start) / 2) + start;
      FindMaxPositionRecursiveTask t1 =
        new FindMaxPositionRecursiveTask(data, start, halfWay);
      t1.fork(); // queue left half of task
      FindMaxPositionRecursiveTask t2 =
        new FindMaxPositionRecursiveTask(data, halfWay, end);
      int position2 = t2.compute(); // work on right half of task
      int position1 = t1.join();    // wait for queued task to be complete
      // out of the position in two subsection which is greater?
      if (data[position1] > data[position2]) {
        return position1;
      } else if (data[position1] < data[position2]) {
        return position2;
      } else {
        return position1 < position2 ? position1 : position2;
      }
    }
  }
}
```

To begin the application, we reuse the `RecursiveAction` example to create a large array and initialize it using Fork/Join. After initializing the array with random values, we reuse the `ForkJoinPool` with our `RecursiveTask` to find the position with the greatest value:

```
public static void main(String[] args) {
  int[] data = new int[10_000_000];
  ForkJoinPool fjPool = new ForkJoinPool();
  RandomInitRecursiveAction action =
    new RandomInitRecursiveAction(data, 0, data.length);
  fjPool.invoke(action);
  // new code begins here
  FindMaxPositionRecursiveTask task =
   new FindMaxPositionRecursiveTask(data, 0, data.length);
  Integer position = fjPool.invoke(task);
  System.out.println("Position: " + position + ", value: " + data[position]);
}
```

Notice that a value is returned by the call to invoke when using a `RecursiveTask`.

If your application will repeatedly submit tasks to a ***`ForkJoinPool`****, then you should reuse a single* ***`ForkJoinPool`*** *instance and avoid the overhead involved in creating a new instance.*

Embarrassingly Parallel

A problem or task is said to be embarrassingly parallel if little or no additional work is required to solve the problem in a parallel fashion. Sometimes, solving a problem in parallel adds so much more overhead that the problem can be solved faster serially. The `RandomInitRecursiveAction` example, which initializes an array to random values, has no additional overhead because what happens when processing one subsection of an array has no bearing on the processing of another subsection. Technically, there is a small

amount of overhead even in the `RandomInitRecursiveAction`; the Fork/Join Framework and the `if` statement that determines whether the problem should be subdivided both introduce some overhead. Be aware that it can be difficult to get performance gains that scale with the number of CPUs you have. Typically, four CPUs will result in less than a 4× speedup when moving from a serial to a parallel solution.

The `FindMaxPositionRecursiveTask` example, which finds the largest value in an array, does introduce a small additional amount of work because you must compare the result from each subsection and determine which is greater. This is only a small amount, however, and adds little overhead. Some tasks may introduce so much additional work that any advantage of using parallel processing is eliminated (the task runs slower than serial execution). If you find yourself performing a lot of processing after calling `join()`, then you should benchmark your application to determine if there is a performance benefit to using parallel processing. Be aware that performance benefits might only be seen with a certain number of CPUs. A task might run on one CPU in 5 seconds, on two CPUs in 6 seconds, and on four CPUs in 3.5 seconds.

The Fork/Join Framework is designed to have minimal overhead as long as you don't over-subdivide your tasks and the amount of work required to join results can be kept small. A good example of a task that incurs additional overhead but still benefits from Fork/Join is array sorting. When you split an array into two 766halves and sort each half separately, you then have to combine the two sorted arrays, as shown in the following example:

```
public class SortRecursiveAction extends RecursiveAction {
  private static final int THRESHOLD = 1000;
  private int[] data;
  private int start;
  private int end;

  public SortRecursiveAction(int[] data, int start, int end) {
    this.data = data;
    this.start = start;
    this.end = end;
  }

  @Override
  protected void compute() {
    if (end - start <= THRESHOLD) {
      Arrays.sort(data, start, end);
    } else {
    int halfWay = ((end - start) / 2) + start;
    SortRecursiveAction a1 =
      new SortRecursiveAction(data, start, halfWay);
    SortRecursiveAction a2 =
      new SortRecursiveAction(data, halfWay, end);
    invokeAll(a1, a2);  // shortcut for fork() & join()
      if(data[halfWay-1] <= data[halfWay]) {
        return; // already sorted
      }
        // merging of sorted subsections begins here
        int[] temp = new int[end - start];
        int s1 = start, s2 = halfWay, d = 0;
        while(s1 < halfWay && s2 < end) {
          if(data[s1] < data[s2]) {
          temp[d++] = data[s1++];
        } else if(data[s1] > data[s2]) {
          temp[d++] = data[s2++];
        } else {
            temp[d++] = data[s1++];
          temp[d++] = data[s2++];
        }
      }
        if(s1 != halfWay) {
        System.arraycopy(data, s1, temp, d, temp.length - d);
```

```
        } else if(s2 != end) {
          System.arraycopy(data, s2, temp, d, temp.length - d);
        }
        System.arraycopy(temp, 0, data, start, temp.length);
      }
    }
  }
```

In the previous example, everything after the call to `invokeAll` is related to merging two sorted subsections of an array into a single larger sorted subsection.

Because Java applications are portable, the system running your application may not have the hardware resources required to see a performance benefit. Always perform testing to determine which problem and hardware combinations see performance increases when using Fork/Join.

CERTIFICATION OBJECTIVE

Parallel Streams (OCP Objective 10.6)

10.6 Use parallel Streams including reduction, decomposition, merging processes, pipelines and performance.

Parallel streams are designed for problems you can divide and conquer, just like the problems described in the previous section. In fact, parallel streams are implemented with Fork/Join tasks under the covers, so you're essentially doing the same thing: that is, splitting up a problem into subtasks that can be executed on separate threads and then joining them back together to produce a result.

Unlike the code you write to take advantage of the Fork/Join Framework, parallel stream code is relatively easy to write. If you think writing `RecursiveActions` and `RecursiveTasks` is tricky (we do too!), you'll be pleased with how much easier parallel streams can be. That said, there are several gotchas to watch out for when using parallel streams, so even though the code is easier to write, you need to pay close attention.

And, like we said about using the Fork/Join Framework, just because you can use parallel streams to solve a problem doesn't always mean you should. The same concerns apply: the effort to create tasks that can run in threads can add enough overhead that using parallel streams is sometimes slower than using sequential streams. We'll do testing as we work through some parallel stream code and check our results as we go to make sure we get the performance gains we expect.

How to Make a Parallel Stream Pipeline

In Chapter 9, "Streams," we made a stream parallel by calling the `parallel()` method on the stream, like this:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int sum = nums.stream()
          .parallel()          // make the stream parallel
          .mapToInt(n -> n)
          .sum();
System.out.println("Sum is: " + sum);
```

Here, we take a list of numbers, stream them, map each `Integer` to an `int`, and sum. To make this very simple stream pipeline parallel, we simply call the `parallel()` method on the stream.

Let's take a peek at how the tasks created by this parallel stream get split up and handled by workers in the `ForkJoinPool`. We can do that by adding a `peek()` into the pipeline and then printing the name of the thread handling each `Integer` in the stream:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int sum = nums.stream()
          .parallel()          // make the stream parallel
          .peek(i ->           // print the thread for the worker
            System.out.println(i + ": "
              + Thread.currentThread().getName()))
          .mapToInt(n -> n)
          .sum();
System.out.println("Sum is: " + sum);
```

The output (on our computer) is

```
7: main
5: ForkJoinPool.commonPool-worker-5
8: ForkJoinPool.commonPool-worker-4
9: ForkJoinPool.commonPool-worker-2
3: ForkJoinPool.commonPool-worker-1
1: ForkJoinPool.commonPool-worker-6
10: ForkJoinPool.commonPool-worker-7
2: ForkJoinPool.commonPool-worker-3
6: main
4: ForkJoinPool.commonPool-worker-5
Sum is: 55
```

You will likely see different output (although the same final sum) because your computer may have a different number of cores, and the tasks will likely get split up differently. We ran this code on a computer with eight cores, and you can see that the computer used all eight. We might not actually want the computer to use all eight cores (especially for this simple sum), so we can tell the computer 769exactly how many workers to use by creating a custom `ForkJoinPool` and then submitting a task to the pool for execution:

```
ForkJoinPool fjp = new ForkJoinPool(2);
try {
   int sum =
      fjp.submit(              // returns a Future (FutureTask)
         () -> nums.stream()   // a Callable (value returning task)
            .parallel()        // make the stream parallel
            .peek(i ->
                 System.out.println(i + ": " +
                     Thread.currentThread().getName()))
            .mapToInt(n -> n)
            .sum()
      ).get();                 // from Future; get() waits for
                               // computation to complete and
                               // gets the result
  System.out.println("FJP with 2 workers, sum is: " + sum);
} catch (Exception e) {
   System.out.println("Error executing stream sum");
   e.printStackTrace();
}
```

Here is our output now:

```
7: ForkJoinPool-1-worker-1
6: ForkJoinPool-1-worker-1
9: ForkJoinPool-1-worker-1
3: ForkJoinPool-1-worker-0
10: ForkJoinPool-1-worker-1
5: ForkJoinPool-1-worker-0
8: ForkJoinPool-1-worker-1
4: ForkJoinPool-1-worker-0
2: ForkJoinPool-1-worker-1
1: ForkJoinPool-1-worker-0
FJP with 2 workers, sum is: 55
```

We get the same sum, 55, but now we're using only two workers to do it. This is friendlier to the computer, which may want some cores to do other things while this task is running.

When we call the `submit()` method of the `ForkJoinPool`, we use this method:

```
<T> ForkJoinTask<T> submit(Callable<T> task)
```

and pass a lambda expression for the `Callable`. `Callable` is a functional interface, which is why we can express an instance of a class implementing that interface with the lambda expression. The functional method in `Callable` is `call()`, which "computes a result" (thus, the lambda we are supplying for the `Callable` is a `Supplier`: it takes no arguments and supplies a result).

You've seen how to create a parallel stream with `parallel()`; you can also combine the methods `stream()` and `parallel()` into one method, `parallelStream()`, like this:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int sum = nums.parallelStream()      // make a parallel stream
          .peek(i -> System.out.println(i + ": "
                + Thread.currentThread().getName()))
          .mapToInt(n -> n)
          .sum();
System.out.println("Sum is: " + sum);
```

When we run this code (similar to the first version of the code above), we get the following output:

```
2: ForkJoinPool.commonPool-worker-3
3: ForkJoinPool.commonPool-worker-1
9: ForkJoinPool.commonPool-worker-2
7: main
8: ForkJoinPool.commonPool-worker-4
1: ForkJoinPool.commonPool-worker-6
5: ForkJoinPool.commonPool-worker-5
10: ForkJoinPool.commonPool-worker-7
6: ForkJoinPool.commonPool-worker-1
4: ForkJoinPool.commonPool-worker-3
Sum is: 55
```

Notice the ordering of the processing: the ordering is totally different from the first example above! When summing numbers, it doesn't matter in what order they are summed; we'll get the same result every time. In Chapter 9, you saw that if you display the result of a parallel stream pipeline, the ordering is not guaranteed, and you can see that mixed-up ordering in the output here, too.

You can check to see if a stream is parallel with the `isParallel()` method. This might come in handy if someone hands you a stream and you're not sure. For instance, the following code checks to see if `numsStream` is parallel:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
Stream<Integer> numsStream =       // create a parallel stream
    nums.parallelStream();
// Later...
System.out.println("Is numsStream a parallel stream?? "
    + numsStream.isParallel());
```

You should see the output:

```
Is numsStream a parallel stream?? true
```

What if you have a parallel stream and you want to make it not parallel (i.e., sequential)? You can do that with the `sequential()` method:

```
Stream<Integer> numsStreamSeq =    // make stream sequential
    numsStream.sequential();
System.out.println("Is numsStreamSeq a parallel stream?? "
    + numsStreamSeq.isParallel());
```

and get the output:

```
Is numsStreamSeq a parallel stream?? false
```

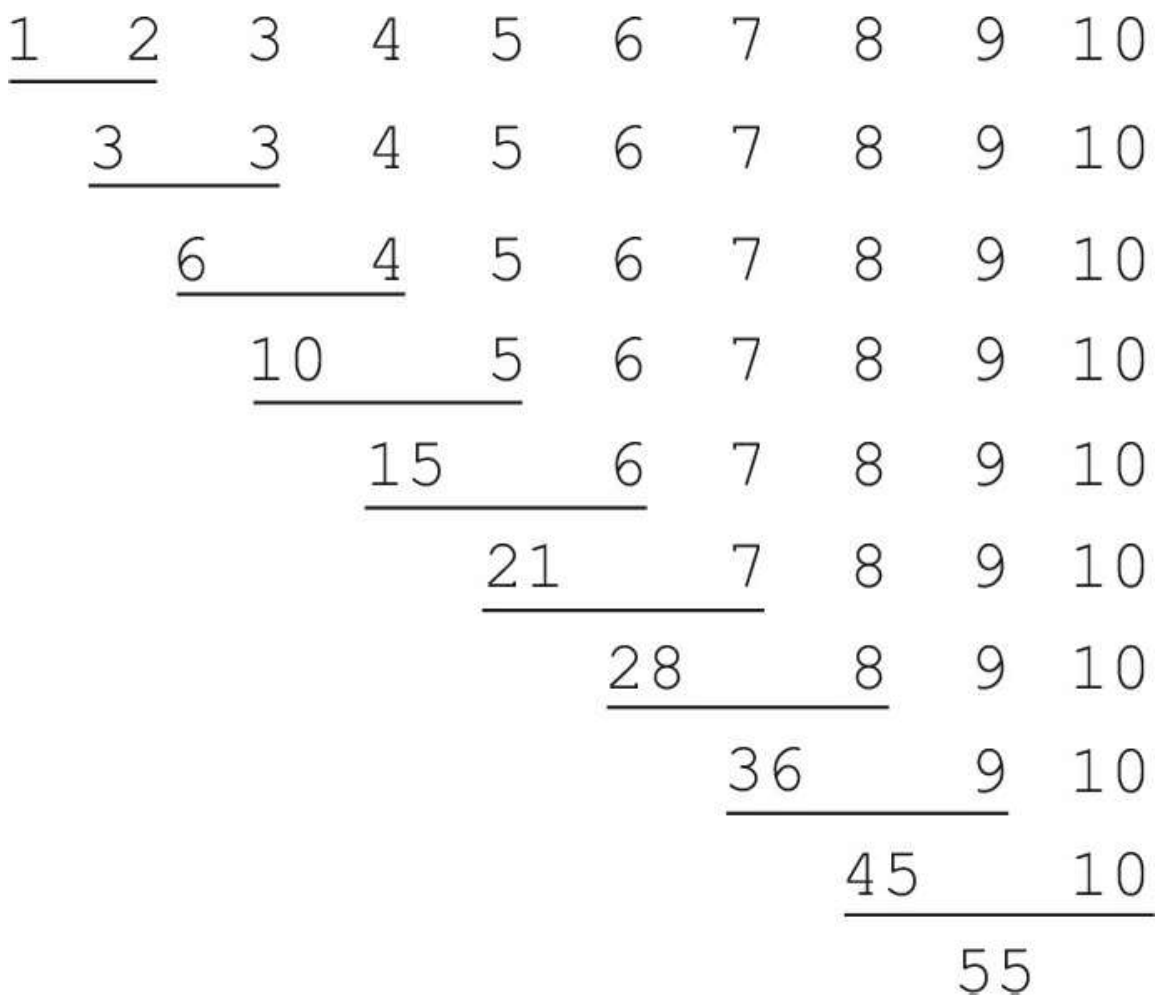
Note here that `parallel()`, `isParallel()`, and `sequential()` are methods of `BaseStream` (which is the superinterface of `Stream`), and `parallelStream()` is a method of the `Collection` interface. Table 11-1 summarizes the methods related to parallel streams that you are expected to know for the exam, which we'll cover in this chapter.

TABLE 11-1 Methods Related to Parallel Streams

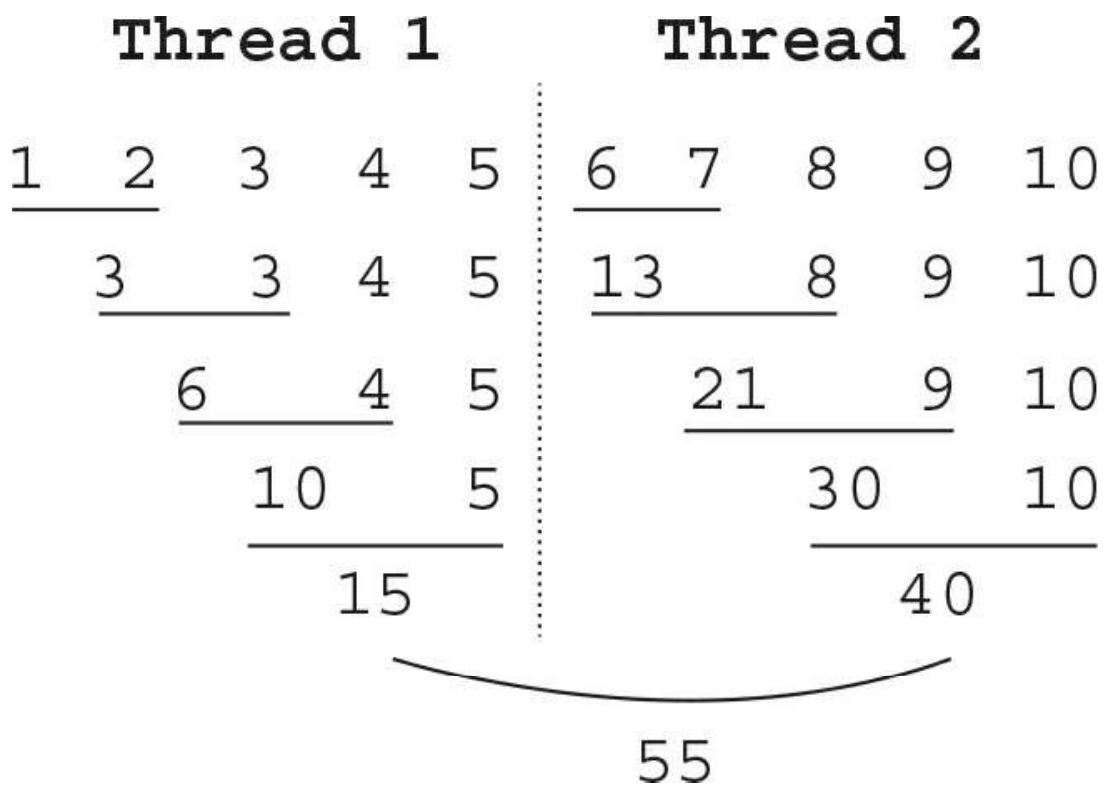
Method	Of Interface	Description
`parallel()`	`BaseStream`	Creates a parallel stream from a stream
`parallelStream()`	`Collection`	Creates a parallel stream from the Collection source
`isParallel()`	`BaseStream`	Returns true if the stream is parallel (that is, if the stream would execute in parallel when a terminal operation is executed)
`sequential()`	`BaseStream`	Returns a sequential stream
`unordered()`	`BaseStream`	Creates an unordered stream
`forEachOrdered()`	`Stream`	Consumes elements from a stream and performs an action on those elements, in the encounter order of the original stream if that stream has an order

Embarrassingly Parallel, Take Two (with Parallel Streams)

Earlier we talked about how some problems are "embarrassingly parallel": that is, they're easily split up into independent pieces that can be computed separately and then combined. These are precisely the kinds of problems that work well with parallel streams. The sum example we've been using in this section is a great example of an embarrassingly parallel problem: we can split up the stream into subsections, compute the sum of the subsection, and then combine the sums from each to make one total sum. We visualized this process in Chapter 9 with two images. The first illustrates how we compute a sum with a sequential stream:



while the second image illustrates how we can compute a sum with a parallel stream that is using two workers:



As you can see in the second illustration, the stream can be split in two and each sum computed completely independently from the other and then combined at the end. Again, this is precisely the kind of problem that works well with parallel streams.

We can summarize the conditions that make for successful parallel streams as follows: a problem is most suitable for parallel streams if the pipeline operations are *stateless,* the reduction operation used to compute the result is *associative* and *stateless,* and the stream is *unordered*.

Let's take a look at each of these.

Associative Operations

We talked about associative operations earlier in Chapter 9. Recall from that chapter that `sum()` is an example of an associative operation. That is, we can compute the sum of a + b, and then add c, or we can compute b + c, and then add a, and get the same result.

Some operations are definitely not associative. We discovered in Chapter 9 that computing the average of a stream of numbers is not associative. Recall that when we implemented our own average reduction operation, we got an incorrect result. Using the built-in `average()` method solved the problem in that chapter.

Because parallel streams are split up for processing in unpredictable ways, operations that aren't associative will probably fail. Just to reinforce how important associativity is in properly reducing a stream, let's review how computing the average fails when we don't use the built-in `average()` method:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
OptionalDouble avg = nums
    .parallelStream()                    // make a parallel stream
    .mapToDouble(n -> n)                 // make a stream of doubles
    .reduce((d1, d2) -> (d1 + d2) / 2); // reduce with (bad) average
avg.ifPresent((a) ->
    System.out.println("Average of parallel stream with reduce: " + a));
OptionalDouble avg2 = nums
    .parallelStream()                    // make a parallel stream
    .mapToDouble(n -> n)                 // make a stream of doubles
    .average();                          // reduce with built-in average
avg2.ifPresent((a) ->
    System.out.println("Average of parallel stream with average: " + a));
```

In the first stream pipeline, we make the stream parallel, then we map to a stream of `doubles` so we can compute the average and return a `double` value; then we reduce using our own reduction function, which sums two numbers from the stream and divides by 2.

In the second stream pipeline, we do the same except we reduce with the built-in `average()` stream method, which is implemented in a way to produce the correct result (accounting for the inherent nonassociativity of the average function).

We get the output:

```
Average of parallel stream with reduce: 5.125  // wrong!
Average of parallel stream with average: 5.5   // correct!
```

The point is, just as with sequential stream reductions, parallel stream reductions need to be associative. This is even more important for parallel streams given that we don't know how the system will split up and then recombine the stream results.

Stateless Operations (and Streams)

A stateless operation in a stream pipeline is an operation that does not depend on the context in which it's operating. Before we talk more about what stateless operations are, let's talk about what they are not.

There are two main ways you can create a state*ful* stream pipeline. The first is with *side*

effects. We talked about this in Chapter 9, too; side effects occur when your result creates or depends on changes to state in the pipeline.

Side Effects and Parallel Streams We already said not to ever modify the source of a stream from within the stream pipeline, so you haven't been—right? Right. That's one way you can create side effects, but you know not to do that.

Another way you can create side effects is to modify the field of an object. In sequential stream pipelines, you can get away with these kinds of side effects, and you may recall from Chapter 9 how we created a list of DVDs from a file input stream. However, in parallel stream pipelines, these kinds of side effects will fail unless the objects you're modifying from within the pipeline are synchronized (either via a synchronized accessor method or if the objects are a concurrent data type).

Here's an example: We're streaming integers between 0 and 50 (noninclusive), filtering to find integers divisible by 10 and then summing those. We're also attempting to count the integers as we go by updating the field of an object. Although we aren't allowed to update a plain variable counter from within a lambda (you'll get a compiler error if you try to do this), we are allowed to update the field of an object, as we're doing here:

```
public static void stateful() {
  class Count {                  // an object to hold our counter
    int counter = 0;
  }
  Count count = new Count();     // create an instance
  IntStream stream =             // generate a stream of integers, 0-49
    IntStream.range(0, 50);
  int sum = stream
    .parallel()                  // make the stream parallel
    .filter(i -> {               // filter the stream
      if (i % 10 == 0) {         // …only count numbers divisible by 10
        count.counter++;         // …there should be 5!
        return true;
      }
      return false;
    })
  .sum();                        // sum up the integers
  System.out.printf("sum: " + sum + ", count: " + count.counter);
}
```

The first time we ran this code, we got:

```
sum: 100, count: 5
```

The next time:

```
sum: 100, count: 4
```

Hmmm. We get the right sum (the sum doesn't depend on the counter and doesn't depend on the ordering), but the counter is not always correct (try a bigger number than 50 in the range if this code doesn't produce different `counter` values for you). That's because, when we are using a parallel stream, multiple threads are accessing the `Count` object to

modify the counter field and `Count` is not thread-safe.

We could use a synchronized object to store our counter and that would solve the problem, but it would also defeat the purpose of using parallel streams in this example.

We can fix the code above by removing the count in the parallel stream pipeline and computing it separately from the sum.

Stateful Operations in the Stream Pipeline Another way we can create stateful stream pipelines is with stateful stream operations. A stateful stream operation is one that requires some knowledge about the stream in order to operate. Take a look at the following stream pipeline:

```
IntStream stream = IntStream.range(0, 10);
long sum = stream.limit(5).sum();
System.out.println("Sum is: " + sum);
```

In this code, we are creating a stream of ten `int`s, limiting the stream to the first five of those `int`s (0–4) and summing those. The result, as you'd expect, is 10:

```
Sum is: 10
```

Now think about this operation, `limit(5)`. This is a stateful operation. Why? Because it requires context: the stream has to keep some intermediate state to know when it has five items and can stop streaming from the source (that is, short circuit the stream). So adding `parallel()` to this stream will not improve performance and might even hurt performance because now that state has to be synchronized across threads.

Let's test this and see what performance we get. We'll create a stream of 100 million `int`s, limit the stream to the first 5 `int`s, and sum, like we just did above. We'll also time the operation:

```
final int SIZE = 100_000_000;
final int LIMIT = 5;
long sum = 0, startTime, endTime, duration;
```

```
IntStream stream = IntStream.range(0, SIZE);
startTime = Instant.now().toEpochMilli();
sum = stream
        .limit(LIMIT)
        .sum();
endTime = Instant.now().toEpochMilli();
duration = endTime - startTime;
System.out.println("Items summed in " + duration
   + " milliseconds; sum is: " + sum);
```

When we run this on our machine (eight cores), we get

```
Items summed in 29 milliseconds; sum is: 10
```

Running it several times takes 28 or 29 milliseconds each time.

Now, let's make this a parallel stream and see what we get:

```
IntStream stream = IntStream.range(0, SIZE);
startTime = Instant.now().toEpochMilli();
sum = stream
        .parallel()
        .limit(LIMIT)
        .sum();
endTime = Instant.now().toEpochMilli();
duration = endTime - startTime;
System.out.println("Items summed in " + duration
   + " milliseconds; sum is: " + sum);
```

Running this, we get

```
Items summed in 34 milliseconds; sum is: 10
```

The performance is worse! Repeated runs yield running times of between 33 and 36

milliseconds each time. Increasing the `SIZE` to 400 million yields similar results.

One thing we should consider here is that the overhead of creating eight threads might be contributing to the performance problem. To really test that a parallel pipeline can hurt (or, at least, not help) when using `limit()`, we should try our experiment with a custom `ForkJoinPool` and set the number of threads ourselves. Let's do that.

The code is similar to what you've seen before: we create a custom `ForkJoinPool`, submit the task to the `ForkJoinPool`, get the result from the `FutureTask` that's returned, and time the whole thing. We've bumped up the 777size of the stream to 400 million `int`s, but we're still limiting to 5 `int`s for the sum. For this initial test, we've commented out the call to `parallel()` in the stream pipeline, so our first test will be on a sequential stream with one thread in the `ForkJoinPool`:

```
final int SIZE = 400_000_000;
final int LIMIT = 5;
long sum = 0, startTime, endTime, duration;
ForkJoinPool fjp = new ForkJoinPool(1);    // Limit FJP to 1 thread
IntStream stream = IntStream.range(0, SIZE);
try {
  startTime = Instant.now().toEpochMilli();
  sum =
    fjp.submit(
      () -> stream
            //.parallel()                  // test sequential first
            .limit(LIMIT)
            .sum()
    ).get();
  endTime = Instant.now().toEpochMilli();
  duration = endTime - startTime;
  System.out.println("FJP Stream data summed in "
      + duration + " milliseconds; sum is: " + sum);
} catch (Exception e) {
  System.out.println("Error executing stream sum");
  e.printStackTrace();
}
```

When we ran this code, we got

```
FJP Stream data summed in 35 milliseconds; sum is: 10
```

Now let's make the stream pipeline parallel and increase the number of threads to two:

```
ForkJoinPool fjp = new ForkJoinPool(2);  // Now use 2 threads
IntStream stream = IntStream.range(0, SIZE);
try {
  startTime = Instant.now().toEpochMilli();
  sum =
    fjp.submit(
      () -> stream
              .parallel()                // make the stream parallel
              .limit(LIMIT)
              .sum()
    ).get();
  endTime = Instant.now().toEpochMilli();
  duration = endTime - startTime;
  System.out.println("FJP Stream data summed in "
      + duration + " milliseconds; sum is: " + sum);
} catch (Exception e) {
  System.out.println("Error executing stream sum");
  e.printStackTrace();
}
```

Run it again and we get

```
FJP Stream data summed in 36 milliseconds; sum is: 10
```

Now using parallel streams is not that much slower than running it sequential, but there's still no benefit to using parallel streams; our results from this test indicate that whether we use two threads or eight, the parallel pipeline runs no faster than the sequential pipeline.

Run some more experiments yourself. Change the number of workers in the `ForkJoinPool`; see what happens. Change the `LIMIT` to a much larger number, like 500,000, and see what happens. More than likely, you'll find that the parallel stream

pipeline runs the same as or more slowly than the sequential stream pipeline for this operation, which is what we'd expect.

We've talked about a couple of ways that stream pipelines are stateful: that is, stream pipelines are stateful if we create side effects (modifying an object as we process the pipeline) or if we use a stateful stream operation, like `limit()`. Other stateful stream operations might be fairly obvious to you; they are `skip()`, `distinct()`, and `sorted()`. How do you know which stream operations are stateful? The documentation describing these methods says so. And if you think about it, each of these operations requires some knowledge about the stream in order for the stream to operate.

Stream operations that are not necessarily stateful include `map()`, `filter()`, and `reduce()` (among others), so there's a lot we can do with streams that is stateless, although, as with all things related to concurrency, we need to be careful. As you saw in the example above, when we tried to count numbers in the stream using a `filter()`, we can make `filter()` stateful by creating a side effect.

Just to revisit quickly a simple example of a parallel stream using `map()`, `filter()`, and `reduce()` stateless operations, here's a slightly modified take on our original example:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
long sum = nums.stream()
          .parallel()              // make the stream parallel
          .mapToInt(n -> n)        // map from Integer to int
          .filter(i ->             // filter the evens
             i % 2 == 0 ? true : false)
          .sum();                  // sum the evens
System.out.println("Sum of evens is: " + sum);
```

Run this and you should see the following:

```
Sum of evens is: 30
```

None of these stream operations requires any state in order to operate on the values in the stream pipeline. The map simply converts the type of the stream value from `Integer` to `int`; the filter simply determines if the value is even or odd, and the reduction (`sum()`) sums the values, which can happen in any order because `sum()` is associative, so this operation requires no state either. This is an example of a stateless stream pipeline…well almost.

Hopefully now, you can see the difference between a stateless and a stateful stream pipeline. We've mostly talked about what a stateless stream pipeline is by talking about what a stateful stream pipeline is. The short story is that a stateless stream pipeline is one

that requires no underlying intermediate state to be stored and accessed by the thread(s) in order to execute properly. And that statelessness is what helps make parallel stream pipelines more efficient.

However, we need to do one more thing to make this stream pipeline completely stateless, as you'll see next.

Unordered Streams

Earlier we said a problem is suitable for parallel streams if the operations used to compute the result are associative and stateless and the stream is unordered. We've talked about the first two; what about unordered?

By default, many (but not all!) streams are ordered. That means there is an inherent ordering to the items in the stream. A stream of `int`s created by `range()` and a stream of `Integer`s created from a `List` of `Integer`s are both ordered streams. Intuitively that makes sense; technically, ordering is determined by whether the stream has an `ORDERED` *characteristic*. This is just a bit that is set on the underlying implementation of a stream. There are other characteristics of streams, including `SORTED` and `DISTINCT`, but you don't need to worry too much about characteristics of streams except to know that the characteristics of a stream can affect how that stream performs, especially if you make the stream parallel.

Sometimes we want our streams to retain their order; for instance, if we are mapping stream values from `int`s to `Integer`, filtering to extract the even numbers, and then displaying the results, we might want the ordering of the stream to be maintained so we see the results in order.

However, an ordered stream pipeline will not execute as efficiently in parallel. Again, it comes down to context: an ordered stream has to maintain some state—in this case, the ordering of the stream values—to keep the stream values in order, which adds overhead to the processing.

The characteristics of a stream can be inspected by using a ***`Spliterator`****, which is an object for traversing a source, like a collection or a stream, that can also split up that source for potential parallel processing.* ***`Spliterator`*** *and its characteristics are not on the exam, but if you want to explore how streams (and collections) are traversed and partitioned in more detail, you can study* ***`Spliterator`*** *in depth.*

Here's how you can use a stream's ***`spliterator()`*** *method to determine if that stream is ordered:*

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
Stream<Integer> s = nums.stream();
System.out.println("Stream from List ordered? " +
   s.spliterator().hasCharacteristics(Spliterator.ORDERED));
```

In this case you'll see that, indeed, the stream of Integers we make from a **`List`** *is ordered.*

If a stream is ordered and we process it in parallel, the stream will remain ordered, but at the price of some efficiency. If we don't care about the ordering, as is the case when we are summing numbers, we might as well remove the ordering on the stream in scenarios where we are using a parallel stream pipeline. That way we get the extra performance benefits of working with an unordered stream in a situation where we shouldn't be concerned about the ordering anyway (because if we are, then the problem we're trying to solve probably isn't appropriate for a parallel stream pipeline).

So how do we make sure we're working with an unordered stream? We can explicitly tell the stream to not worry about remaining ordered by calling the `unordered()` stream method. Of course, we should do this *before* we call `parallel()` so we can maximize the efficiency of the parallel processing. Here's how we can modify the previous example to create an unordered stream pipeline:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
long sum = nums.stream()
            .unordered()    // make the stream unordered
            .parallel()
            .mapToInt(n -> n)
            .filter(i -> i % 2 == 0 ? true : false)
            .sum();
System.out.println("Sum of evens is: " + sum);
```

Calling `unordered()` doesn't change the ordering of the stream; it just unsets that `ORDERED` bit so the stream doesn't have to maintain the ordering state. Depending on how the stream is processed in the parallel pipeline, you may find that the final stream (before the reduction) is in the same order as the source, or not. Note that an ordered stream is not the same thing as a sorted stream. Once you've made a stream unordered, there is no way to order it again, except by calling `sorted()`, which makes it sorted *and* ordered but not necessarily in the same order as it was in the original source.

Don't worry too much about ordering for the exam; do remember that ordering has an

implication for performance and make sure to note which collections create ordered streams (e.g., `List`) and which create unordered streams (e.g., `HashSet`) when you call that collection's `stream()` method, so you know the performance implications.

To summarize: to make the most efficient parallel stream, you should:

- Make sure your reductions are associative and stateless.
- Avoid side effects.
- Make sure your pipeline is stateless by avoiding contextual operations such as `limit()`, `skip()`, `distinct()`, and `sorted()`.

forEach() and forEachOrdered()

As you know, `forEach()` is a terminal stream operation that takes a `Consumer` and consumes each item in the stream. We often use `forEach()` to show the values in a stream after a map and/or filter operation, for example:

```
dogs.stream().filter(d -> d.getAge() > 7).forEach(System.out::println);
```

will display any `Dog` in the stream whose age is > 7 in the order in which the values in the stream are encountered.

Imagine we have a `Dog` class and a `Dog` constructor that takes the name and age of a dog. You've seen the `Dog` class often enough that we probably don't have to repeat it. Given that `Dog` class, we can make some dogs, like this:

```
List<Dog> dogs = new ArrayList<>();
Dog aiko = new Dog("aiko", 10);
Dog boi = new Dog("boi", 6);
Dog charis = new Dog("charis", 7);
Dog clover = new Dog("clover", 12);
Dog zooey = new Dog("zooey", 8);
dogs.add(aiko); dogs.add(boi); dogs.add(charis);
dogs.add(clover); dogs.add(zooey);
```

Notice that we've added the dogs in alphabetical order on purpose, so we can watch the ordering of our stream when we use a sequential stream and when we use a parallel stream.

Running the code:

```
dogs.stream().filter(d -> d.getAge() > 7).forEach(System.out::println);
```

gives us the output:

```
aiko is 10 years old
clover is 12 years old
zooey is 8 years old
```

That is, we see all dogs whose age is > 7 in the order in which they are encountered in the stream (which is the order in which they are added to the `List`). This is because, by default, when we stream the `List`, we get a sequential, ordered stream.

Now, as you might expect at this point, when we make this a parallel stream, we get different results:

```
dogs.stream()
        .parallel()
        .filter(d -> d.getAge() > 7)
        .forEach(System.out::println);
```

This code produced the output:

```
zooey is 8 years old
aiko is 10 years old
clover is 12 years old
```

The output is unpredictable; if you run it again, you may get different results. Even though the stream is ordered (since we're streaming a `List`), we see the output in a random order because the stream is parallel, so the dogs can be processed by different threads that may finish at different times, and the `forEach() Consumer` is stateful: it is essentially creating a side effect by outputting data to the console, which is not thread-safe.

We can make sure we see the dogs in the order in which they appear in the original source by using the method `forEachOrdered()` instead:

```
dogs.stream().parallel().filter(d -> d.getAge() > 7)
    .forEachOrdered(System.out::println);      // enforce ordering
```

We can do this because the underlying stream is ordered (it's made from a `List` and its `ORDERED` characteristic is set), and the `forEachOrdered()` method will make sure the items are seen in the same order as they are encountered in the ordered stream. (Note that if we call `unordered()` on the stream, then the order will not be guaranteed!)

Of course, this takes a bit of overhead to perform, right? As we discussed earlier, an ordered stream is going to be less efficient when processed in parallel than an unordered stream. However, there may be times when you want to maintain the ordering of the stream when processing in parallel and you're willing to sacrifice a bit of efficiency.

Typically, you won't end up using `forEachOrdered()` much in the real world; you're sacrificing performance, and usually you don't want to see the results of a big data operation on a parallel stream; you want to collect the results in a new data structure or compute some final result like a sum. However, you will see `forEachOrdered()` on the exam, so make sure you understand how it works.

Also, note that `forEach()` (and `forEachOrdered()`) as well as `peek()` are stream operations that are designed for side effects. That is, they consume elements from the stream: `peek()` typically creates a side effect (say, printing the stream element passing through) and then passes the values on unchanged to the stream; `forEach()` consumes the stream values and produces a result (it's a terminal operation), and typically that result is output to the console or saves each item to the field of an object. So take care when using `forEach()`, `forEachOrdered()`, and `peek()` with parallel streams, remembering that stream pipelines with side effects (even just printing to the console) can change how the stream pipeline operates, and reduce the performance of parallel streams.

A Quick Word About findAny()

You might remember from Chapter 9 that we used the `findAny()` stream operation to find any value in the stream pipeline that matched a `filter()`. For instance, we can use `findAny()` to find any even `int` in a stream like this:

```
IntStream nums = IntStream.range(0, 20);
OptionalInt any = nums
        .filter(i ->            // filter the evens
          i % 2 == 0 ? true : false)
        .findAny();             // find any even int
any.ifPresent(i -> System.out.println("Any even is: " + i));
```

With a sequential stream, `findAny()` will likely return the first value in the stream, 0, every time, even though it's not guaranteed.

Now, let's parallelize this stream and see what happens. We'll add a `peek()`, so we can see the thread workers as they work on the problem:

```
IntStream nums = IntStream.range(0, 20);
OptionalInt any = nums
        .parallel()          // make the stream parallel
                             // peek at the thread name
        .peek(i -> System.out.println(i + ": "
           + Thread.currentThread().getName()))
           .filter(i ->      // filter the evens
           i % 2 == 0 ? true : false)
        .findAny();          // find any even int
any.ifPresent(i -> System.out.println("Any even is: " + i));
```

We run this and get

```
12: main
1: ForkJoinPool.commonPool-worker-6
5: ForkJoinPool.commonPool-worker-7
2: ForkJoinPool.commonPool-worker-3
6: ForkJoinPool.commonPool-worker-1
8: ForkJoinPool.commonPool-worker-5
17: ForkJoinPool.commonPool-worker-2
16: ForkJoinPool.commonPool-worker-4
Any even is: 12
```

Remember, our computer has eight cores, so this stream pipeline has been split up into eight workers, each tackling part of the stream. The `findAny()` method is short circuiting, so even though we still have 12 more values in the stream to process (since the stream has 20 values), we stop as soon as we find the first even number. In this run, we had several threads with even numbers: the main worker, worker-1, worker-4, and worker-5. It just so happens that the main thread probably got done first, and so as soon as that even number was found, everything else stopped, and that result, 12, was returned.

Run the code again, and you'll likely get a different answer.

This example illustrates that `findAny()` really does find any result, particularly when

you're working with a parallel stream pipeline.

A Parallel Stream Implementation of a RecursiveTask

Let's write code to sum an array of `int`s with a `ForkJoinPool RecursiveTask`, and compare that code with a parallel stream (which, remember, uses `ForkJoinPool` under the covers). We'll also implement it with a plain-old `for` loop, so you can see not only how the code itself compares, but also how the performance compares. Much of this code should look familiar to you by now.

```
/*
* Sum numbers in an array of SIZE random numbers
* from 1 to MAX if number > NUM
*/
public class SumRecursiveTask extends RecursiveTask<Long> {
   public static final int SIZE = 400_000_000;
   public static final int THRESHOLD = 1000;
   public static final int MAX = 10;      // array of numbers, 1-10
   public static final int NUM = 5;       // sum numbers > 5
   private int[] data;
   private int start;
   private int end;
   public SumRecursiveTask(int[] data, int start, int end) {
      this.data = data;
      this.start = start;
      this.end = end;
   }
   @Override
   protected Long compute() {
      long tempSum = 0;
      if (end - start <= THRESHOLD) {
         for (int i = start; i < end; i++) {
            if (data[i] > NUM) {
            vtempSum += data[i];
            }
         }
         return tempSum;
      } else {
         int halfWay = ((end - start) / 2) + start;
         SumRecursiveTask t1 = new SumRecursiveTask(data, start, halfWay);
         SumRecursiveTask t2 = new SumRecursiveTask(data, halfWay, end);
         t1.fork();                // queue left half of task
         long sum2 = t2.compute(); // compute right half
         long sum1 = t1.join();    // compute left and join
         return sum1 + sum2;
      }
   }
   public static void main(String[] args) {
      int[] data2sum = new int[SIZE];
      long sum = 0, startTime, endTime, duration;
      // create an array of random numbers between 1 and MAX
      for (int i = 0; i < SIZE; i++) {
        data2sum[i] = ThreadLocalRandom.current().nextInt(MAX) + 1;
      }
      startTime = Instant.now().toEpochMilli();
      // sum numbers with plain old for loop
      for (int i = 0; i < data2sum.length; i++) {
         if (data2sum[i] > NUM) {
```

```
            sum = sum + data2sum[i];
        }
    }
    endTime = Instant.now().toEpochMilli();
    duration = endTime - startTime;
    System.out.println("Summed with for loop in " + duration
        + " milliseconds; sum is: " + sum);

    // sum numbers with ResursiveTask
    ForkJoinPool fjp = new ForkJoinPool();
    SumRecursiveTask action =
        new SumRecursiveTask(data2sum, 0, data2sum.length);
    startTime = Instant.now().toEpochMilli();
    sum = fjp.invoke(action);
    endTime = Instant.now().toEpochMilli();
    duration = endTime - startTime;
    System.out.println("Summed with recursive task in "
        + duration + " milliseconds; sum is: " + sum);

    // sum numbers with a parallel stream
    IntStream stream2sum = IntStream.of(data2sum);
    startTime = Instant.now().toEpochMilli();
    sum =
        stream2sum
            .unordered()
            .parallel()
            .filter(i -> i > NUM)
            .sum();
    endTime = Instant.now().toEpochMilli();
    duration = endTime - startTime;
    System.out.println("Stream data summed in " + duration
        + " milliseconds; sum is: " + sum);

    // sum numbers with a parallel stream, limiting workers
    ForkJoinPool fjp2 = new ForkJoinPool(4);
    IntStream stream2sum2 = IntStream.of(data2sum);
    try {
        startTime = Instant.now().toEpochMilli();
        sum =
            fjp2.submit(
                    () -> stream2sum2
                            .unordered()
                            .parallel()
                            .filter(i -> i > NUM)
                            .sum()
                    ).get();
        endTime = Instant.now().toEpochMilli();
        duration = endTime - startTime;
        System.out.println("FJP4 Stream data summed in "
         + duration + " milliseconds; sum is: " + sum);
    } catch (Exception e) {
```

```
            System.out.println("Error executing stream average");
            e.printStackTrace();
        }
    }
}
```

This code first generates a large array of random numbers between 1 and 10 and then computes the sum of all numbers > 5 four different times (from the same data, so we should get the same answer each time): first, using a plain-old `for` loop; second, using a `RecursiveTask`, third, using a parallel stream on the default `ForkJoinPool` (that is, using all eight cores of our machine); and finally, using a parallel stream on a custom `ForkJoinPool` with four workers (using four cores).

Here are our results:

```
Summed with for loop in 287 milliseconds; sum is: 399980957
Summed with recursive task in 267 milliseconds; sum is: 399980957
Stream data summed in 118 milliseconds; sum is: 399980957
FJP4 Stream data summed in 136 milliseconds; sum is: 399980957
```

We ran it again and this time we got

```
Summed with for loop in 292 milliseconds; sum is: 399927370
Summed with recursive task in 196 milliseconds; sum is: 399927370
Stream data summed in 184 milliseconds; sum is: 399927370
FJP4 Stream data summed in 138 milliseconds; sum is: 399927370
```

In the first run, the parallel stream running on all eight cores was the winner, slightly faster than the parallel stream running on four cores. Both parallel streams were far superior to the `RecursiveTask` and the `for` loop.

In the second run, this time the parallel stream running on four cores was the clear winner, with the `RecursiveTask` and the parallel stream running on eight cores about the same, both much faster than the `for` loop.

As you can see, your results will vary depending on the solution you choose as well as your underlying machine architecture. If you get an `OutOfMemoryError` when you run this, try reducing the `SIZE`.

Reducing Parallel Streams with reduce()

Let's say you want to reduce your stream to a value, but none of the built-in stream reduction methods (like `sum()`) suffice. You can build your own custom reduction with the `reduce()` function, as you saw in Chapter 9.

How about building a custom reduction that multiplies all the elements of a stream? It's a bit like computing the sum, except we're multiplying instead. This is known as computing the *product* (as compared to the sum). Before we write this reduction, we should check a few things. First, remember that a reduction produces an `Optional` value unless we provide an identity.

Recall the type signature of `reduce()`:

```
T reduce(T identity, BinaryOperator<T> accumulator)
```

Although the identity is optional, providing one is a good idea because it allows us to get a result (rather than an `Optional` result), and that identity value is also used to make the first result in the stream pipeline, and we want to make sure the correct identity is being used in our custom reduction.

For the sum, the identity value is 0 because any number added to 0 is that number. For the product, the identity is 1 because any number multiplied by 1 is that number. So, we'll use 1. That's easy enough.

The accumulator is a `BinaryOperator` that takes two values and produces one value. Remember that an `Operator` produces a value of the same type as its arguments. Our `BinaryOperator` is simple; it just takes two numbers and multiples them together, returning a number:

```
(i1, i2) -> i1 * i2
```

Another thing we should check is that our accumulator function is associative. Remember the trouble we ran into when we tried to reduce using a custom average function? We didn't get a correct result because our average function is not associative, so we had to use the built-in `average()` reduction method instead.

Is our product function associative? It is. That is, we can multiply a * b and then by c and get the same answer as when we multiply b * c and then by a.

Another quick check: Is our accumulator stateless? That is, does it rely on any additional state in the stream to be computed? If it's not stateless, then we could run into trouble; either we'll potentially get incorrect results or we'll drastically reduce the performance of the parallel stream (or both!). In this case, our reduction function is, indeed, stateless. There is no state needed in order to properly multiply two values from the stream and produce a result.

Okay! We've got our identity, and we've got our associative, stateless reduction function. Now we can write the code to reduce a parallel stream to the product of all values in the

stream:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int mult = nums.stream()
      .unordered()                        // unordered for efficiency
      .parallel()                         // make the stream parallel
      .reduce(1,  (i1, i2) -> i1 * i2);   // reduce to the product
System.out.println("Product reduction: " + mult);
```

Here, we start with a `List` of `Integer` values and create a stream. We call `unordered()` on that stream so we can get additional efficiency from parallel processing —and this works great, because just like addition, computing the product does not depend on the order of the values. We then make the stream parallel and call the `reduce()` terminal operation method, passing in our identity, 1, and our product accumulator. Here is the result we get:

```
Product reduction: 3628800
```

That's the correct answer (phew!). Running the code again several times produces the same result each time. It looks like we got the code correct.

As an exercise, you can try this reduction on a much larger stream of numbers and time the performance, comparing the parallel version with the sequential version. See what results you get!

Collecting Values from a Parallel Stream

As we've said before, when processing data with streams, you will most likely want to reduce the stream to a single value, like a sum or a product or a count, or you might want to collect results into a new collection with `collect()` (which is a reduction, too).

Consider again our `ArrayList` of dogs we used above, containing aiko, boi, charis, clover, and zooey.

If you forget that you aren't supposed to create side effects from a parallel stream pipeline, you might do something like this to collect dogs who are older than 7 into a new `List` collection:

```
List<Dog> dogsOlderThan7 = new ArrayList<>();
long count = dogs.stream()                  // stream the dogs
    .unordered()                            // make the stream unordered
    .parallel()                             // make the stream parallel
    .filter(d -> d.getAge() > 7)            // filter the dogs
    .peek(d -> dogsOlderThan7.add(d))       // save… with a side effect
    .count();
System.out.println("Dogs older than 7, via side effect: " + dogsOlderThan7);
```

Here, we're initializing a new `List`, `dogsOlderThan7`; we're streaming our original dogs `ArrayList`, making it unordered, making it parallel, filtering for dogs older than 7, and then making the mistake of saving those dogs older than 7 to our new `List` using a side effect from within a `peek()` lambda expression. We then terminate the pipeline with the `count()` operation.

The first time we ran this we got the following:

```
Dogs older than 7, via side effect: [null, aiko is 10 years old]
```

Hmm, definitely a problem there! Running it again, we got this:

```
Dogs older than 7, via side effect: [null, null, clover is 12 years old]
```

Clearly, as you should know well by now, this is not the way to collect results from a stream pipeline. The reason this fails is because we have multiple threads trying to access the `List dogsOlderThan7` at the same time and `dogsOlderThan7` is not thread-safe.

We could fix this code by using a synchronized list, like this:

```
List<Dog> dogsOlderThan7 =
    Collections.synchronizedList(new ArrayList<>());
```

Now, the `List` is thread-safe. The order of the dogs in the resulting `List` may not be the same as in the source, but at least you'll get the right set of dogs. However, you're also sacrificing some performance because you're forcing a synchronization of all the threads attempting to write to the synchronized `List`.

A better way to collect values from a parallel stream pipeline is to use `Collectors`, as we did in Chapter 9 with sequential streams. Here's how:

```
List<Dog> dogsOlderThan7 =
    dogs.stream()                          // stream the dogs
        .unordered()                       // make the stream unordered
        .parallel()                        // make the stream parallel
        .filter(d -> d.getAge() > 7)       // filter dogs older than 7
        .collect(Collectors.toList());     // collect older dogs into a List
System.out.println("Dogs older than 7: " + dogsOlderThan7);
```

When we run this code, we see the output:

```
Dogs older than 7: [aiko is 10 years old, clover is 12 years old,
zooey is 8 years old]
```

Now you might be thinking, "Didn't you just say that the `List` is not thread-safe? We're still using a `List` in the `collect()` method with `Collectors.toList()`, so how is this working?"

Good question. It turns out that the way `collect()` works under the covers is that each worker processing a piece of the stream collects its data into its own collection. Worker 1 will create a `List` of dogs older than 7; worker 2 will create a `List` of dogs older than 7; and so on. Each of these `Lists` is separate, built from the pieces of the stream that each worker got when the stream was split up across the parallel workers. At the end, once each thread is complete, the separate `Lists` are merged together, in a thread-safe manner, to create one final `List` of dogs.

`Lists` are inherently ordered, so we will get a reduction in efficiency unless we make sure the stream is unordered, which we've done by adding a call to `unordered()` at the beginning of the pipeline. (Of course, this means the dogs may not be in the same order as they were in the source, but you expected that, right?). You can also collect values from a parallel stream with `toSet()` and `toMap()`, and a method `toConcurrentMap()` is there for additional efficiency since merging maps is expensive.

As we mentioned in Chapter 9, you can use the `collect()` method with your own supplier, accumulator, and combiner functions if you need a way to collect that isn't provided for by the `Collectors` class:

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

If you use a custom collector, just like with the custom reduction, you'll need to make sure that your accumulator is associative and stateless. In addition, your combiner should be stateless and thread-safe. Typically, the combiner will be adding values to a new collection, so using a concurrent collection here is a good idea.

CERTIFICATION SUMMARY

This chapter covered the required concurrency knowledge you'll need to apply on the certification exam. The `java.util.concurrent` package and its subpackages form a high-level multithreading framework in Java. You should become familiar with threading basics before attempting to apply the Java concurrency libraries, but once you learn `java.util.concurrent`, you may never extend `Thread` again.

`Callable`s and `Executors` (and their underlying thread pools) form the basis of a high-level alternative to creating new `Threads` directly. As the trend of adding more CPU cores continues, knowing how to get Java to make use of them all concurrently could put you on easy street. The high-level APIs provided by `java.util.concurrent` help you create efficient multithreaded applications while eliminating the need to use low-level threading APIs such as `wait()`, `notify()`, and `synchronized`, which can be a source of hard-to-detect bugs.

When using an `Executor`, you will commonly create a `Callable` implementation to represent the work that needs to be executed concurrently. A `Runnable` can be used for the same purpose, but a `Callable` leverages generics to allow a generic return type from its `call` method. `Executor` or `ExecutorService` instances with predefined behavior can be obtained by calling one of the factory methods in the `Executors` class like so: `ExecutorService es = Executors.newFixedThreadPool(100);`.

Once you obtain an `ExecutorService`, you submit a task in the form of a `Runnable` or `Callable`, or a collection of `Callable` instances to the `ExecutorService` using one of the `execute`, `submit`, `invokeAny`, or `invokeAll` methods. An `ExecutorService` can be held onto during the entire life of your application if needed, but once it is no longer needed, it should be terminated using the `shutdown` and `shutdownNow` methods.

We looked at the Fork/Join Framework, which supplies a highly specialized type of `Executor`. Use the Fork/Join Framework when the work you would typically put in a `Callable` can be split into multiple units of work. The purpose of the Fork/Join Framework is to decrease the amount of time it takes to solve a problem by leveraging the additional CPUs in a system. You should only run a single Fork/Join task at a time in an application, because the goal of the framework is to allow a single task to consume all available CPU resources in order to be solved as quickly as possible. In most cases, the effort of splitting a single task into multiple tasks that can be operated on by the underlying Fork/Join threads will introduce additional overhead. Don't assume that applying Fork/Join will grant you a performance benefit for all problems. The overhead involved may be large enough that any benefit of applying the framework is offset.

When applying the Fork/Join Framework, first subclass either `RecursiveTask` (if a return result is desired) or `RecursiveAction`. Within one of these `ForkJoinTask` subclasses, you must implement the `compute` method. The `compute()` method is where you divide the work of a task into parts and then call the `fork` and `join` methods or the `invokeAll` method. To execute the task, create a `ForkJoinPool` instance with `ForkJoinPool pool = new ForkJoinPool();` and submit the `RecursiveTask` or `RecursiveAction` to the pool with the `pool.invoke(task)` method. Although the Fork/Join API itself is not that large, creating a correct and efficient implementation of a `ForkJoinTask` can be challenging.

Java 8 added parallel streams to make the Fork/Join Framework easier to use. Parallel streams are built on top of the Fork/Join Framework and allow you to split a task into parts more easily than when using the Fork/Join API directly. Simply create a stream and call `parallel()`, and your stream pipeline will be split into tasks that execute in separate threads. However, as you saw, you need to be aware of the potential issues with parallel streams and make sure you are using them to solve the appropriate kinds of problems.

We learned about the `java.util.concurrent` collections. There are three categories of collections: copy-on-write collections, concurrent collections, and blocking queues. The copy-on-write and concurrent collections are similar in use to the traditional `java.util` collections but are designed to be used efficiently in a thread-safe fashion. The copy-on-write collections (`CopyOnWriteArrayList` and `CopyOnWriteArraySet`) should be used for read-heavy scenarios. When attempting to loop through all the elements in one of the copy-on-write collections, always use an `Iterator`. The concurrent collections included

- `ConcurrentHashMap`
- `ConcurrentLinkedDeque`
- `ConcurrentLinkedQueue`
- `ConcurrentSkipListMap`
- `ConcurrentSkipListSet`

These collections are meant to be used concurrently without requiring locking. Remember that iterators of these five concurrent collections are weakly consistent. `ConcurrentHashMap` and `ConcurrentSkipListMap` are `ConcurrentMap` implementations that add atomic `putIfAbsent`, `remove`, and `replace` methods to the `Map` interface. Seven blocking queue implementations are provided by the `java.util.concurrent` package:

- `ArrayBlockingQueue`
- `LinkedBlockingDeque`
- `LinkedBlockingQueue`
- `PriorityBlockingQueue`

- `DelayQueue`
- `LinkedTransferQueue`
- `SynchronousQueue`

These blocking queues are used to exchange objects between threads—one thread will deposit an object and another thread will retrieve that object. Depending on which queue type is used, the parameters used to create the queue, and the method being called, an insert or a removal operation may block until it can be completed successfully. In Java 7, the `LinkedTransferQueue` class was added and acts as a superset of several blocking queue types; you should prefer it when possible.

Another way to coordinate threads is to use a `CyclicBarrier`. A `CyclicBarrier` creates a barrier where all threads must wait until all participating threads reach that barrier; once they do, then the threads can continue. You can have an optional `Runnable` run before the threads continue; the last thread to reach the barrier is the thread that's used to run that `Runnable`.

The `java.util.concurrent.atomic` and `java.util.concurrent.locks` packages contain additional utility classes you might consider using in concurrent applications. The `java.util.concurrent.atomic` package supplies thread-safe classes that are similar to the traditional wrapper classes (such as `java.lang.Integer`) but with methods that support atomic modifications. The `java.util.concurrent.locks.Lock` interface and supporting classes enable you to create highly customized locking behaviors that are more flexible than traditional object monitor locking (the `synchronized` keyword).

TWO-MINUTE DRILL

Here are some of the key points from the certification objectives in this chapter.

Apply Atomic Variables and Locks (OCP Objective 10.3)

- The `java.util.concurrent.atomic` package provides classes that are similar to volatile fields (changes to an atomic object's value will be correctly read by other threads without the need for synchronized code blocks in your code).
- The atomic classes provide a `compareAndSet` method that is used to validate that an atomic variable's value will only be changed if it matches an expected value.
- The atomic classes provide several convenience methods such as `addAndGet` that will loop repeatedly until a `compareAndSet` succeeds.
- The `java.util.concurrent.locks` package contains a locking mechanism that is an alternative to synchronized methods and blocks. You get greater flexibility

at the cost of a more verbose syntax (such as having to manually call `lock.unlock()` and having an automatic release of a synchronization monitor at the end of a synchronized code block).

- The `ReentrantLock` class provides the basic Lock implementation. Commonly used methods are `lock()`, `unlock()`, `isLocked()`, and `tryLock()`. Calling `lock()` increments a counter and `unlock()` decrements the counter. A thread can only obtain the lock when the counter is zero.
- The `ReentrantReadWriteLock` class provides a `ReadWriteLock` implementation that supports a read lock (obtained by calling `readLock()`) and a write lock (obtained by calling `writeLock()`).

Use java.util.concurrent Collections (OCP Objective 10.4)

- Copy-on-write collections work well when there are more reads than writes because they make a new copy of the collection for each write. When looping through a copy-on-write collection, use an iterator (remember, `for-each` loops use an iterator).
- None of the concurrent collections make the elements stored in the collection thread-safe—just the collection itself.
- `ConcurrentHashMap`, `ConcurrentSkipListMap`, and `ConcurrentSkipListSet` should be preferred over synchronizing with the more traditional collections.
- `ConcurrentHashMap` and `ConcurrentSkipListMap` are `ConcurrentMap` implementations that enhance a standard Map by adding atomic operations that validate the presence and value of an element before performing an operation: `putIfAbsent(K key, V value)`, `remove(Object key, Object value)`, `replace(K key, V value)`, and `replace(K key, V oldValue, V newValue)`.
- Blocking queues are used to exchange objects between threads. Blocking queues will block (hence the name) when you call certain operations, such as calling `take()` when there are no elements to take. There are seven different blocking queues that have slightly different behaviors; you should be able to identify the behavior of each type.

Blocking Queue	Description
`ArrayBlockingQueue`	A FIFO (first-in-first-out) queue in which the head of the queue is the oldest element and the tail is the newest. An `int` parameter to the constructor limits the size of the queue (it is a bounded queue).
`LinkedBlockingDeque`	Similar to `LinkedBlockingQueue`, except it is a double-ended queue (deque). Instead of only supporting FIFO operations, you can remove from the head or tail of the queue.
`LinkedBlockingQueue`	A FIFO queue in which the head of the queue is the oldest element and the tail is the newest. An optional `int` parameter to the constructor limits the size of the queue (it can be bounded or unbounded).
`PriorityBlockingQueue`	An unbounded queue that orders elements using `Comparable` or `Comparator`. The head of the queue is the lowest value.
`DelayQueue`	An unbounded queue of `java.util.concurrent.Delayed` instances. Objects can only be taken once their delay has expired. The head of the queue is the object that expired first.
`LinkedTransferQueue`	Added in Java 7. An unbounded FIFO queue that supports the features of a `ConcurrentLinkedQueue`, `SynchronousQueue`, and `LinkedBlockingQueue`.
`SynchronousQueue`	A blocking queue with no capacity. An `insert` operation blocks until another thread executes a remove operation. A `remove` operation blocks until another thread executes an `insert` operation.

- Some blocking queues are bounded, meaning they have an upper bound on the number of elements that can be added, and a thread calling `put(e)` may block until space becomes available.
- `CyclicBarrier` creates a barrier at which threads must wait until all participating threads reach that barrier. Once all of the threads have reached the barrier, they can continue running. You can use `CyclicBarrier` to coordinate threads so that an action occurs only after another action is complete or to manage data in Collections that are not thread-safe.
- `CyclicBarrier` takes the number of threads that can wait at the barrier and an optional `Runnable` that is run after all threads reach the barrier, but before they continue execution. The last thread to reach the barrier is used to run this `Runnable`.

Use Executors and ThreadPools (OCP Objective 10.1)

- An `Executor` is used to submit a task for execution without being coupled to how or when the task is executed. Basically, it creates an abstraction that can be used in place of explicit thread creation and execution.
- An `ExecutorService` is an enhanced `Executor` that provides additional functionality, such as the ability to execute a `Callable` instance and to shut down (nondaemon threads in an `Executor` may keep the JVM running after your main method returns).
- The `Callable` interface is similar to the `Runnable` interface, but adds the ability to return a result from its `call` method and can optionally throw an exception.
- The `Executors` (plural) class provides factory methods that can be used to construct `ExecutorService` instances, for example: `ExecutorService ex = Executors.newFixedThreadPool(4);`.

Use the Parallel Fork/Join Framework (OCP Objective 10.5)

- Fork/Join enables work stealing among worker threads in order to keep all CPUs utilized and to increase the performance of highly parallelizable tasks.
- A pool of worker threads of type `ForkJoinWorkerThread` is created when you create a `new ForkJoinPool()`. By default, one thread per CPU is created.
- To minimize the overhead of creating new threads, you should create a single Fork/Join pool in an application and reuse it for all recursive tasks.
- A Fork/Join task represents a large problem to solve (often involving a collection or array).
- When executed by a `ForkJoinPool`, the Fork/Join task will subdivide itself into Fork/Join tasks that represent smaller segments of the problem to be solved.

- A Fork/Join task is a subclass of the `ForkJoinTask` class, either `RecursiveAction` or `RecursiveTask`.
- Extend `RecursiveTask` when the `compute()` method must return a value, and extend `RecursiveAction` when the return type is void.
- When writing a `ForkJoinTask` implementation's `compute()` method, always call `fork()` before `join()` or use one of the `invokeAll()` methods instead of calling `fork()` and `join()`.
- You do not need to shut down a Fork/Join pool before exiting your application because the threads in a Fork/Join pool typically operate in daemon mode.

Use Parallel Streams Including Reduction, Decomposition, Merging Processes, Pipelines, and Performance (OCP Objective 10.6)

- Parallel streams are built on top of the Fork/Join pool.
- Parallel streams provide an easier syntax for creating tasks in the Fork/Join pool.
- You can use the default Fork/Join pool, or create a custom pool and submit tasks expressed as parallel streams via a `Callable`.
- Parallel streams split the stream into subtasks that represent portions of the problem to be solved. Each subtask solution is then combined to produce a final result for the terminal operation of the parallel stream.
- Create a parallel stream by calling `parallel()` on a stream object or `parallelStream()` on a `Collection` object.
- Make a parallel stream sequential again by calling the `sequential()` method.
- Test to see if a stream is parallel with the `isParallel()` method.
- Parallel stream pipelines should be stateless, and for optimum performance, parallel streams should be unordered. Reduction operations on parallel streams should be associative and stateless.
- Stateful parallel stream pipelines will either create an error or unexpected results.
- Nonassociative reductions on streams will produce unexpected results.
- Just like sequential streams, parallel streams can have multiple intermediate operations and must have one terminal operation to produce a result.
- Test your parallel streams to verify you are getting the performance benefits you expect. The performance overhead of creating threads can be greater than the performance gain of a parallel stream in some situations.
- Stateful stream operations, such as `distinct()`, `limit()`, `skip()`, and `sorted()`, will limit the performance of your parallel streams.
- Collect results from a parallel stream pipeline with `collect()`, just as we did with

sequential streams. The collecting happens in a thread-safe way, so you can use `collect()` with `Collectors.toList()`, `toSet()`, and `toMap()` safely.

SELF TEST

The following questions might be some of the hardest in the book. It's just a difficult topic, so don't panic. (We know some Java book authors who didn't do well with these topics and still managed to pass the exam.)

1. The following block of code creates a `CopyOnWriteArrayList`, adds elements to it, and prints the contents:

```
CopyOnWriteArrayList<Integer> cowList = new CopyOnWriteArrayList<>();
cowList.add(4);
cowList.add(2);
Iterator<Integer> it = cowList.iterator();
cowList.add(6);
while(it.hasNext()) {
  System.out.print(it.next() + " ");
}
```

   What is the result?

   A. `6`

   B. `12`

   C. `4 2`

   D. `4 2 6`

   E. Compilation fails

   F. An exception is thrown at runtime

2. Given:

```
CopyOnWriteArrayList<Integer> cowList = new CopyOnWriteArrayList<>();
cowList.add(4);
cowList.add(2);
cowList.add(6);
Iterator<Integer> it = cowList.iterator();
cowList.remove(2);
while(it.hasNext()) {
  System.out.print(it.next() + " ");
}
```

Which shows the output that will be produced?

A. `12`

B. `10`

C. `4 2 6`

D. `4 6`

E. Compilation fails

F. An exception is thrown at runtime

3. Which methods from a `CopyOnWriteArrayList` will cause a new copy of the internal array to be created? (Choose all that apply.)

A. `add`

B. `get`

C. `iterator`

D. `remove`

4. Given:

```
ArrayBlockingQueue<Integer> abq = new ArrayBlockingQueue<>(10);
```

Which operation(s) can block indefinitely? (Choose all that apply.)

A. `abq.add(1);`

B. `abq.offer(1);`

C. `abq.put(1);`

D. `abq.offer(1, 5, TimeUnit.SECONDS);`

5. Given the following code fragment:

```
class SingletonTestDrive {
   public static void main(String args[]) {
      CyclicBarrier barrier = new CyclicBarrier(3, () -> {
         System.out.println(Singleton.INSTANCE.getValue());
      });
      Runnable r = () -> {
         for (int i = 0; i < 100; i++) {
           Singleton.INSTANCE.updateValue();
         }
         try {
            barrier.await();
         } catch (InterruptedException | BrokenBarrierException e) {
             e.printStackTrace();
         }
      };
      Thread t1 = new Thread(r);
      Thread t2 = new Thread(r);
      Thread t3 = new Thread(r);
      t1.start(); t2.start(); t3.start();
      System.out.println("Main thread is complete");
   }
}
enum Singleton {
   INSTANCE;
   int value = 0;
   private void doSomethingWithValue() {
     value = value + 1;
   }
```

```
    public synchronized int updateValue() {
        doSomethingWithValue();
        return value;
    }
        public int getValue() {
            return value;
        }
}
```

What do you expect the output to be, and which thread(s) will be responsible for the output? (Choose all that apply.)

A.

```
Main thread is complete
300
```

Main thread, thread t3

B.

```
300
Main thread is complete
```

thread t3, Main thread

C.

```
Main thread is complete
300
```

OR

```
300
Main thread is complete
```

Main thread, the last thread to reach the barrier

D.

The total value displayed could be any number because the Singleton is not thread-s

```
Main thread is complete
```

Main thread, the last thread to reach the barrier

E.

```
Main thread is complete
```

A BrokenBarrierException
Main thread, thread t1

6. Given:

```
ConcurrentMap<String,Integer> ages = new ConcurrentHashMap<>();
ages.put("John", 23);
```

Which method(s) would delete `John` from the map only if his value was still equal to 23?

A.
```
ages.delete("John", 23);
```

B.
```
ages.deleteIfEquals("John", 23);
```

C.
```
ages.remove("John", 23);
```

D.
```
ages.removeIfEquals("John", 23);
```

7. Which method represents the best approach to generating a random number between 1 and 10 if the method will be called concurrently and repeatedly by multiple threads?

A.
```
public static int randomA() {
    Random r = new Random();
    return r.nextInt(10) + 1;
}
```

B.
```
private static Random sr = new Random();
    public static int randomB() {
    return sr.nextInt(10) + 1;
}
```

C.
```
public static int randomC() {
    int i = (int)(Math.random() * 10 + 1);
    return i;
}
```

D.
```
public static int randomD() {
    ThreadLocalRandom lr = ThreadLocalRandom.current();
    return lr.nextInt(1, 11);
}
```

8. Given:

```
AtomicInteger i = new AtomicInteger();
```

Which atomically increment `i` by 9? (Choose all that apply.)

A. `i.addAndGet(9);`

B. `i.getAndAdd(9);`

C. `i.set(i.get() + 9);`

D. `i.atomicIncrement(9);`

E. `i = i + 9;`

9. Given:

```
public class LeaderBoard {
  private ReadWriteLock rwl = new ReentrantReadWriteLock();
  private List<Integer> highScores = new ArrayList<>();
  public void addScore(Integer score) {
    // position A
    lock.lock();
    try {
      if (highScores.size() < 10) {
        highScores.add(score);
      } else if (highScores.get(highScores.size() - 1) < score) {
        highScores.set(highScores.size() - 1, score);
      } else {
        return;
      }
      Collections.sort(highScores, Collections.reverseOrder());
    } finally {
      lock.unlock();
    }
  }
  public List<Integer> getHighScores() {
    // position B
    lock.lock();
    try {
      return Collections.unmodifiableList(new ArrayList<>(highScores));
    } finally {
      lock.unlock();
    }
  }
}
```

Which block(s) of code best match the behavior of the methods in the `LeaderBoard` class? (Choose all that apply.)

A. `Lock lock = rwl.reentrantLock(); // should be inserted at position A`

B. `Lock lock = rwl.reentrantLcock(); // should be inserted at position B`

C. `Lock lock = rwl.readLock(); // should be inserted at position A`

D. `Lock lock = rwl.readLock(); // should be inserted at position B`

E. `Lock lock = rwl.writeLock(); // should be inserted at position A`

F. `Lock lock = rwl.writeLock(); // should be inserted at position B`

10. Given:

```
ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
rwl.readLock().unlock();
System.out.println("READ-UNLOCK-1");
rwl.readLock().lock();
System.out.println("READ-LOCK-1");
rwl.readLock().lock();
System.out.println("READ-LOCK-2");
rwl.readLock().unlock();
System.out.println("READ-UNLOCK-2");
rwl.writeLock().lock();
System.out.println("WRITE-LOCK-1");
rwl.writeLock().unlock();
System.out.println("WRITE-UNLOCK-1");
```

What is the result?

A. The code will not compile

B. The code will compile and output:

```
READ-UNLOCK-1
READ-LOCK-1
READ-LOCK-2
READ-UNLOCK-2
```

C. The code will compile and output:

```
READ-UNLOCK-1
READ-LOCK-1
READ-LOCK-2
READ-UNLOCK-2
WRITE-LOCK-1
WRITE-UNLOCK-1
```

D. A `java.lang.IllegalMonitorStateException` will be thrown

11. Which class contains factory methods to produce preconfigured `ExecutorService` instances?

A. `Executor`

B. `Executors`

C. `ExecutorService`

D. `ExecutorServiceFactory`

12. Given:

```
private Integer executeTask(ExecutorService service,
                            Callable<Integer> task) {
// insert here
}
```

Which set(s) of lines, when inserted, would correctly use the `ExecutorService` argument to execute the `Callable` and return the `Callable`'s result? (Choose all that apply.)

A.
```
try {
    return service.submit(task);
  } catch (Exception e) {
    return null;
  }
```

B.
```
try {
    return service.execute(task);
  } catch (Exception e) {
    return null;
  }
```

C.
```
try {
    Future<Integer> future = service.submit(task
    return future.get();
  } catch (Exception e) {
    return null;
  }
```

D.
```
try {
    Result<Integer> result = service.submit(task
    return result.get();
  } catch (Exception e) {
    return null;
  }
```

13. Which are true? (Choose all that apply.)

    A. A `Runnable` may return a result but must not throw an `Exception`

    B. A `Runnable` must not return a result nor throw an `Exception`

    C. A `Runnable` must not return a result but may throw an `Exception`

    D. A `Runnable` may return a result and throw an `Exception`

    E. A `Callable` may return a result but must not throw an `Exception`

    F. A `Callable` must not return a result nor throw an `Exception`

    G. A `Callable` must not return a result but may throw an `Exception`

    H. A `Callable` may return a result and throw an `Exception`

14. Given:

```
public class IncrementAction extends RecursiveAction {
  private final int threshold;
  private final int[] myArray;
  private int start;
  private int end;
  public IncrementAction(int[] myArray, int start, int end, int threshold) {
    this.threshold = threshold;
    this.myArray = myArray;
    this.start = start;
    this.end = end;
  }
  @Override
  protected void compute() {
    if (end - start < threshold) {
      for (int i = start; i <= end; i++) {
        myArray[i]++;
      }
    } else {
      int midway = (end - start) / 2 + start;
      IncrementAction a1 = new IncrementAction(myArray, start,
                                               midway, threshold);
      IncrementAction a2 = new IncrementAction(myArray, midway + 1,
                                               end, threshold);
      // insert answer here
    }
  }
}
```

Which line(s), when inserted at the end of the `compute` method, would correctly take the place of separate calls to `fork()` and `join()`? (Choose all that apply.)

A.
```
compute();
```

B.
```
forkAndJoin(a1, a2);
```

C.
```
computeAll(a1, a2);
```

D.
```
invokeAll(a1, a2);
```

15. When writing a `RecursiveTask` subclass, which are true? (Choose all that apply.)

A. `fork()` and `join()` should be called on the same task

B. `fork()` and `compute()` should be called on the same task

C. `compute()` and `join()` should be called on the same task

D. `compute()` should be called before `fork()`

E. `fork()` should be called before `compute()`

F. `join()` should be called after `fork()` but before `compute()`

16. Given the following code fragment:

```
public static void sampleTest() {
    Stream<Integer> nums = Stream.of(10, 5, 3, 2);
    Optional<Integer> result =
        nums
            .parallel()
            .map(n -> n * 10)
            .reduce((n1, n2) -> n1 - n2);
    System.out.println("Result: " + result.get());
}
```

What is the result?

A. 0

B. 40

C. The result is unpredictable

D. Compilation fails

E. An exception is thrown at runtime

17. Given the following code fragment:

```
Stream<List<String>> sDogNames =
  Stream.generate(() ->
                  Arrays.asList("boi", "aiko", "charis", "zooey", "clover"))
        .limit(2).unordered();
sDogNames.parallel()
         .flatMap(s -> s.stream())
         .map(s -> s.toUpperCase())
         .forEach(s -> System.out.print(s + " "));
```

What is the result?

A. `BOI AIKO CHARIS ZOOEY CLOVER BOI AIKO CHARIS ZOOEY CLOVER`

B. Most likely, although not guaranteed: `BOI AIKO CHARIS ZOOEY CLOVER BOI AIKO CHARIS ZOOEY CLOVER`

C. A `ConcurrentModificationException` is thrown

D. Compilation fails

E. An exception other than `ConcurrentModificationException` is thrown at runtime

18. Given the following code fragment:

```
Stream<List<String>> sDogNames2 =
   Arrays.asList(
          Arrays.asList("boi", "aiko", "charis", "zooey", "clover"),
          Arrays.asList("boi", "aiko", "charis", "zooey", "clover"))
         .stream().unordered();
                 sDogNames2.parallel()
          .flatMap(s -> s.stream())
          .map(s -> s.toUpperCase())
          .forEach(s -> System.out.print(s + " "));
```

What is the result?

A. `BOI AIKO CHARIS ZOOEY CLOVER BOI AIKO CHARIS ZOOEY CLOVER`

B. The result is unpredictable

C. A `ConcurrentModificationException` is thrown

D. Compilation fails

E. An exception is thrown at runtime

19. Given the code fragment:

```
List<Integer> myNums = Arrays.asList(1, 2, 3, 4, 5);
OptionalInt aNum = myNums.parallelStream().mapToInt(i -> i * 2).findAny();
aNum.ifPresent(System.out::println);
```

What is the result?

A. 1

B. 2

C. The result is unpredictable; it could be any one of the numbers in the stream

D. A `ConcurrentModificationException` is thrown

E. Compilation fails

F. An exception is thrown at runtime

A SELF TEST ANSWERS

1. ☑ C is correct. The `Iterator` is obtained before 6 is added. As long as the reference to the `Iterator` is maintained, it will only provide access to the values 4 and 2.

   ☒ A, B, D, E, and F are incorrect based on the above. (OCP Objective 10.4)

2. ☑ C is correct. Because the `Iterator` is obtained before `remove()` is invoked, it will reflect all the elements that have been added to the collection.

   ☒ A, B, D, E, and F are incorrect based on the above. (OCP Objective 10.4)

3. ☑ A and D are correct. Of the methods listed, only `add` and `remove` will modify the list and cause a new internal array to be created.

   ☒ B and C are incorrect based on the above. (OCP Objective 10.4)

4. ☑ C is correct. The `add` method will throw an `IllegalStateException` if the queue is full. The two `offer` methods will return false if the queue is full. Only the `put` method will block until space becomes available.

   ☒ A, B, and D are incorrect based on the above. (OCP Objective 10.4)

5. ☑ C is correct; you could see the output in either order because we don't know in advance if threads `t1`, `t2`, and `t3` will complete before the main thread or if the main

thread will complete first. However, in either case, `t1`, `t2`, and `t3` will wait at the barrier until t1, t2, and t3 are all at the barrier, and the last thread to reach the barrier will display the output.

☒ A, B, D, and E are incorrect; A and B are incorrect because of the above. D is incorrect because the `Singleton` is thread-safe; `enum` singletons are guaranteed to be created in a thread-safe manner, and we have synchronized the `updateValue()` method. E is incorrect because it is highly unlikely one of these threads will get stuck or time out and none is accessing any collection that is not thread-safe. (OCP Objective 10.4)

6. ☑ C is correct; it uses the correct syntax.

   ☒ The methods for answers A, B, and D do not exist in a `ConcurrentHashMap`. A traditional `Map` contains a single-argument `remove` method that removes an element based on its key. The `ConcurrentMap` interface (which `ConcurrentHashMap` implements) added the two-argument `remove` method, which takes a key and a value. An element will only be removed from the `Map` if its value matches the second argument. A boolean is returned to indicate if the element was removed. (OCP Objective 10.4)

7. ☑ D is correct. The `ThreadLocalRandom` creates and retrieves `Random` instances that are specific to a thread. You could achieve the same effect prior to Java 7 by using the `java.lang.ThreadLocal` and `java.util.Random` classes, but it would require several lines of code. `Math.random()` is thread-safe, but uses a shared `java.util.Random` instance and can suffer from contention problems.

   ☒ A, B, and C are incorrect based on the above. (OCP Objective 10.1)

8. ☑ A and B are correct. The `addAndGet` and `getAndAdd` both increment the value stored in an `AtomicInteger`.

   ☒ Answer C is not atomic because in between the call to `get` and `set`, the value stored by `i` may have changed. Answer D is invalid because the `atomicIncrement` method is fictional, and answer E is invalid because auto-boxing is not supported for the atomic classes. The difference between the `addAndGet` and `getAndAdd` methods is that the first is a prefix method (++x) and the second is a postfix method (x++). (Objective 10.3)

9. ☑ D and E are correct. The `addScore` method modifies the collection and, therefore, should use a write lock, whereas the `getHighScores` method only reads the collection and should use a read lock.

   ☒ A, B, C, and F are incorrect; they will not behave correctly. (Objective 10.3)

10. ☑ D is correct. A lock counts the number of times it has been locked. Calling lock increments the count, and calling unlock decrements the count. If a call to unlock decreases the count below zero, an exception is thrown.

☒ A, B, and C are incorrect based on the above. (OCP Objective 10.3)

11. ☑ B is correct. `Executor` is the super-interface for `ExecutorService`. You use `Executors` to easily obtain `ExecutorService` instances with predefined threading behavior. If the `Executor` interface does not produce `ExecutorService` instances with the behaviors that you desire, you can always look into using `java.util.concurrent.AbstractExecutorService` or `java.util.concurrent.ThreadPoolExecutor` directly.

    ☒ A, C, and D are incorrect based on the above. (OCP Objective 10.1)

12. ☑ C is correct. When you submit a `Callable` to an `ExecutorService` for execution, you will receive a `Future` as the result. You can use the `Future` to check on the status of the `Callable`'s execution, or just use the `get()` method to block until the result is available.

    ☒ A, B, and D are incorrect based on the above. (OCP Objective 10.1)

13. ☑ B and H are correct. `Runnable` and `Callable` serve similar purposes. `Runnable` has been available in Java since version 1. `Callable` was introduced in Java 5 and serves as a more flexible alternative to `Runnable`. A `Callable` allows a generic return type and permits thrown exceptions, whereas a `Runnable` does not.

    ☒ A, C, D, E, F, and G are incorrect statements. (Objective 10.1)

14. ☑ D is correct. The `invokeAll` method is a var-args method that will fork all Fork/Join tasks, except one that will be invoked directly.

    ☒ A, B, and C are incorrect; they would not correctly complete the Fork/Join process. (OCP Objective 10.5)

15. ☑ A and E are correct. When creating multiple `ForkJoinTask` instances, all tasks except one should be forked first, so they can be picked up by other Fork/Join worker threads. The final task should then be executed within the same thread (typically by calling `compute()`) before calling `join()` on all the forked tasks to await their results. In many cases, calling the methods in the wrong order will not result in any compiler errors, so care must be taken to call the methods in the correct order.

    ☒ B, C, D, and F are incorrect based on the above. (OCP Objective 10.5)

16. ☑ C is correct. The result is unpredictable because the reduction function is not associative and the stream is parallel.

    ☒ A, B, D, and E are incorrect based on the above (OCP Objective 10.6)

17. ☑ B is correct. Because of the `limit(2)`, we will see the dog names twice, most likely in order. The `limit()` forces the stream to maintain state about the source data, so it is likely the stream will retain the original source ordering (although that is not guaranteed) even though it's an unordered parallel stream.

☒ A, C, D, and E are incorrect based on the above (OCP Objective 10.6)

18. ☑ B is correct. Unlike the previous sample question, we have no `limit()` so we have a stateless stream pipeline. The stream is unordered, and the pipeline is executing in parallel, so we cannot predict the ordering of the output. Note that if we were using `forEachOrdered()` instead of `forEach()` as the terminal operation, it is likely, but not guaranteed, that we would see the results in the same ordering as the source.

    ☒ A, C, D, and E are incorrect based on the above (OCP Objective 10.6)

19. ☑ C is correct. For a parallel stream, any of the numbers can be multiplied by 2 and returned by `findAny()`.

    ☒ A, B, D, E, and F are incorrect based on the above (OCP Objective 10.6)