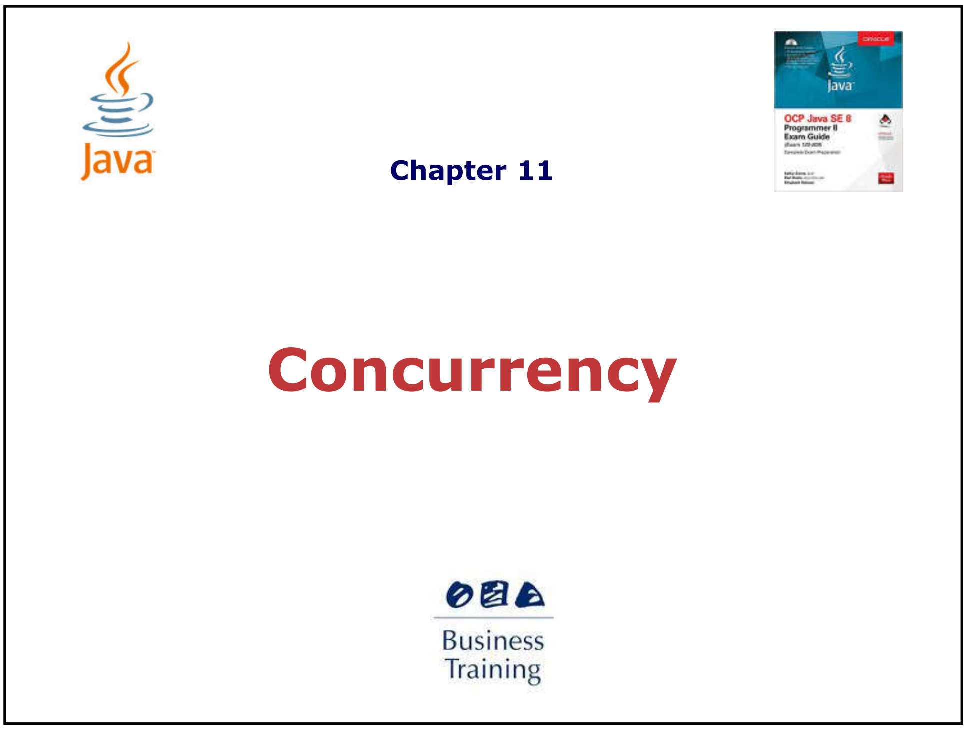
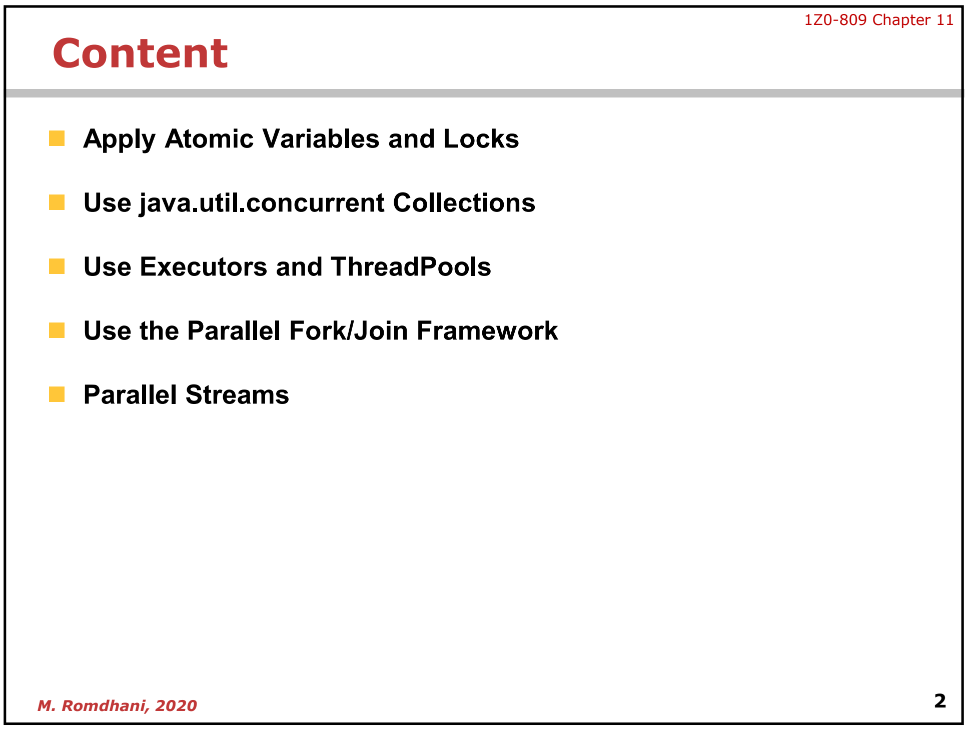


Chapter 11

Concurrency

Business Training

1



1Z0-809 Chapter 11

Content

- Apply Atomic Variables and Locks
- Use java.util.concurrent Collections
- Use Executors and ThreadPools
- Use the Parallel Fork/Join Framework
- Parallel Streams

M. Romdhani, 2020

2

2

Apply Atomic Variables and Locks

3

Atomic Variables

- **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
```

- **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
```

4

Atomic Variables

- **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();
  }
}
```

Locks vs ReetrantLock

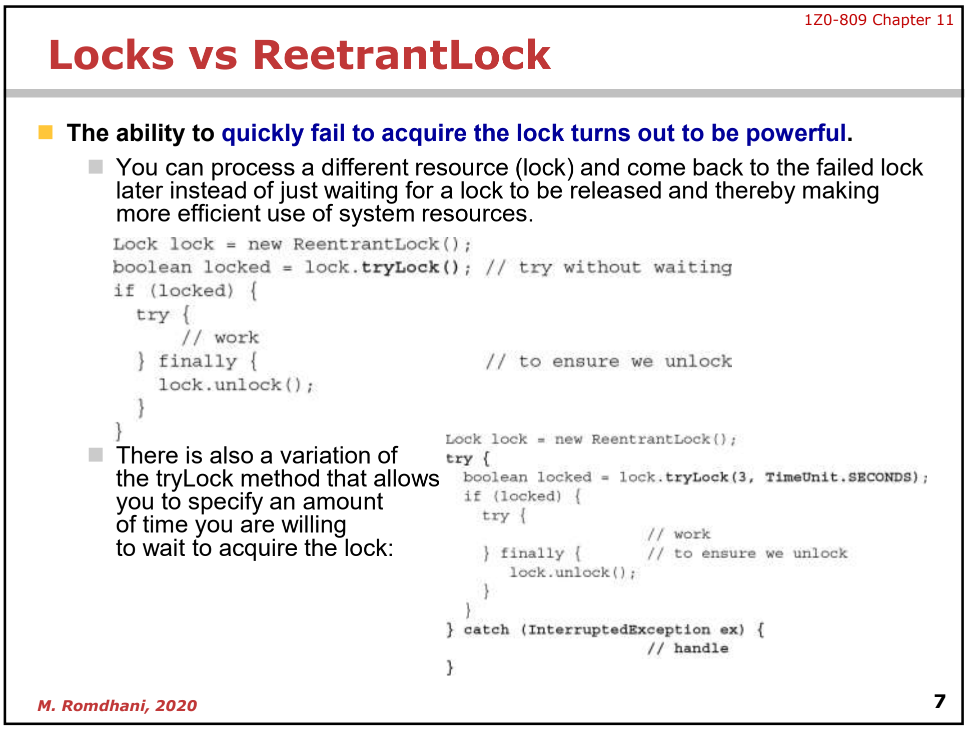
- **To demonstrate the use of Lock, we will first duplicate the functionality of a basic traditional synchronized block.**

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

- ReentrantLock
  - 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.



1Z0-809 Chapter 11

Locks vs ReetrantLock

- **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.

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

  - 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:

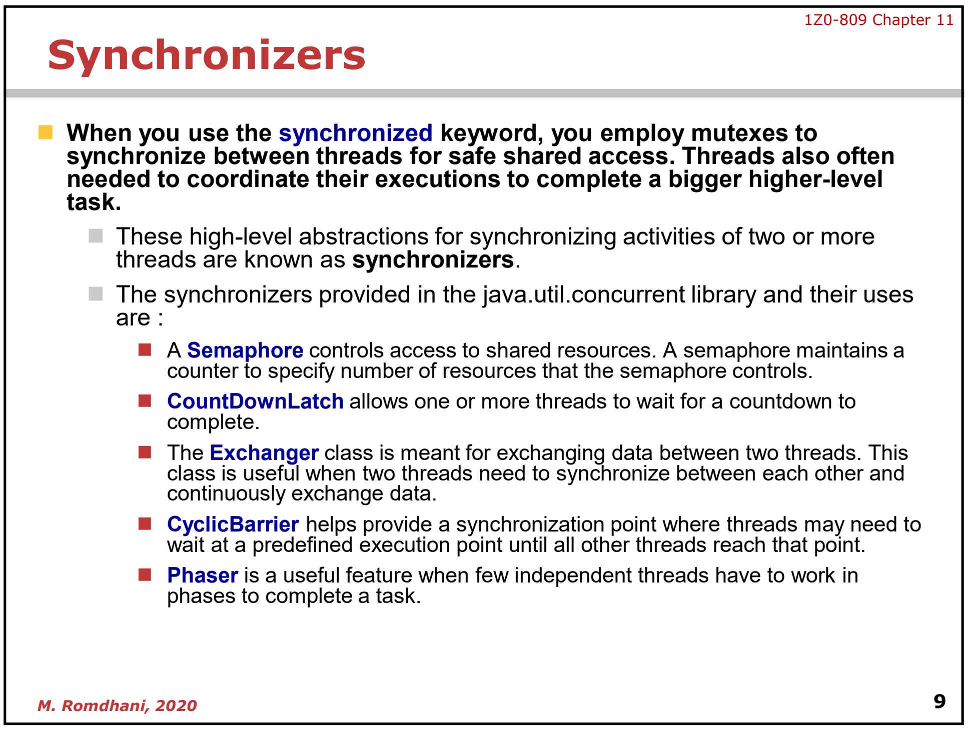
```
Lock lock = new ReentrantLock();
try {
  boolean locked = lock.tryLock(3, TimeUnit.SECONDS);
  if (locked) {
    try {
                    // work
    } finally {     // to ensure we unlock
      lock.unlock();
    }
  }
} catch (InterruptedException ex) {
                    // handle
}
```

M. Romdhani, 2020 7

7

Use java.util.concurrent Collections

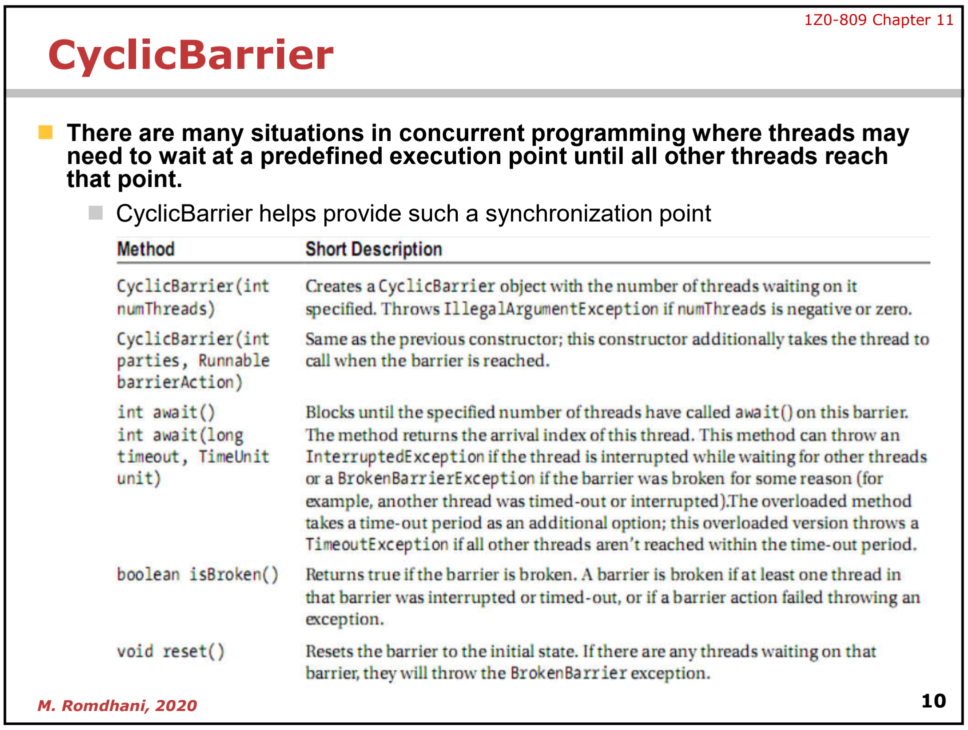
8



Synchronizers

- **When you use the synchronized keyword, you employ mutexes to synchronize between threads for safe shared access. Threads also often needed to coordinate their executions to complete a bigger higher-level task.**
  - These high-level abstractions for synchronizing activities of two or more threads are known as **synchronizers**.
  - The synchronizers provided in the java.util.concurrent library and their uses are :
    - A **Semaphore** controls access to shared resources. A semaphore maintains a counter to specify number of resources that the semaphore controls.
    - **CountDownLatch** allows one or more threads to wait for a countdown to complete.
    - The **Exchanger** class is meant for exchanging data between two threads. This class is useful when two threads need to synchronize between each other and continuously exchange data.
    - **CyclicBarrier** helps provide a synchronization point where threads may need to wait at a predefined execution point until all other threads reach that point.
    - **Phaser** is a useful feature when few independent threads have to work in phases to complete a task.

9

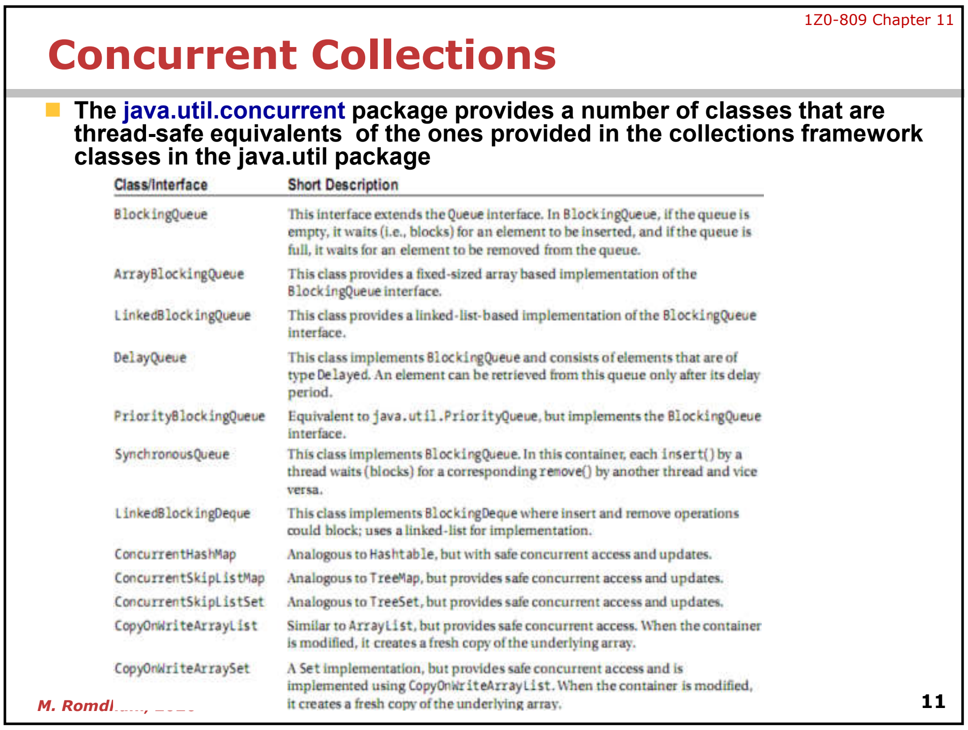


CyclicBarrier

- **There are many situations in concurrent programming where threads may need to wait at a predefined execution point until all other threads reach that point.**
  - CyclicBarrier helps provide such a synchronization point

Method	Short Description
`CyclicBarrier(int numThreads)`	Creates a `CyclicBarrier` object with the number of threads waiting on it specified. Throws `IllegalArgumentException` if `numThreads` is negative or zero.
`CyclicBarrier(int parties, Runnable barrierAction)`	Same as the previous constructor; this constructor additionally takes the thread to call when the barrier is reached.
`int await()` `int await(long timeout, TimeUnit unit)`	Blocks until the specified number of threads have called `await()` on this barrier. The method returns the arrival index of this thread. This method can throw an `InterruptedException` if the thread is interrupted while waiting for other threads or a `BrokenBarrierException` if the barrier was broken for some reason (for example, another thread was timed-out or interrupted).The overloaded method takes a time-out period as an additional option; this overloaded version throws a `TimeoutException` if all other threads aren't reached within the time-out period.
`boolean isBroken()`	Returns true if the barrier is broken. A barrier is broken if at least one thread in that barrier was interrupted or timed-out, or if a barrier action failed throwing an exception.
`void reset()`	Resets the barrier to the initial state. If there are any threads waiting on that barrier, they will throw the `BrokenBarrier` exception.

10



1Z0-809 Chapter 11

Concurrent Collections

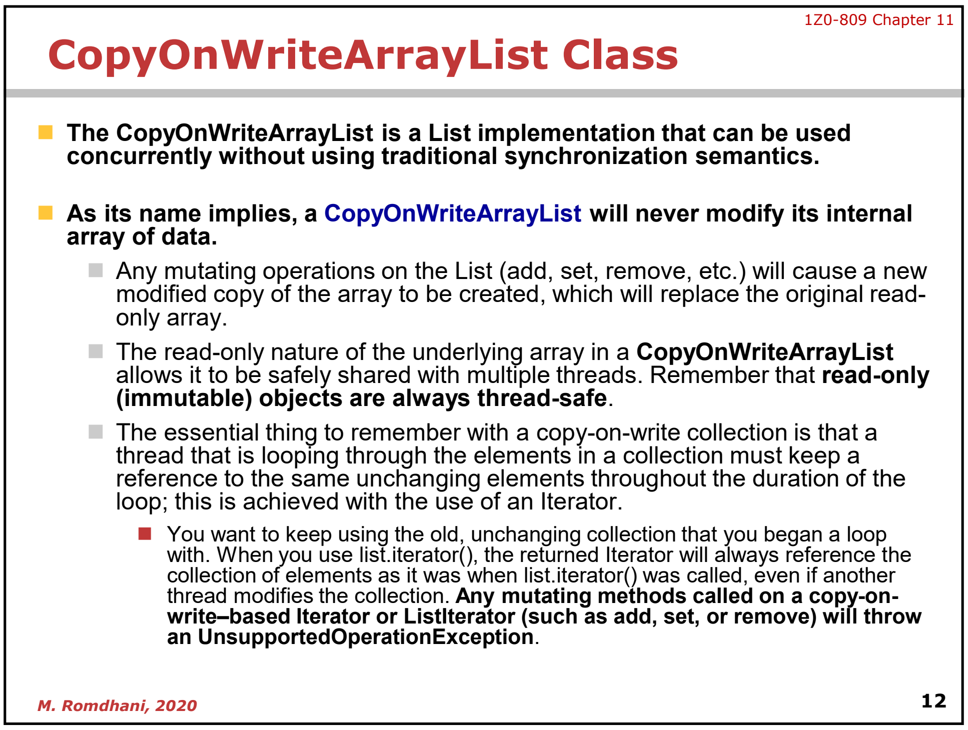
- **The java.util.concurrent package provides a number of classes that are thread-safe equivalents of the ones provided in the collections framework classes in the java.util package**

Class/Interface	Short Description
BlockingQueue	This interface extends the Queue interface. In BlockingQueue, if the queue is empty, it waits (i.e., blocks) for an element to be inserted, and if the queue is full, it waits for an element to be removed from the queue.
ArrayBlockingQueue	This class provides a fixed-sized array based implementation of the BlockingQueue interface.
LinkedBlockingQueue	This class provides a linked-list-based implementation of the BlockingQueue interface.
DelayQueue	This class implements BlockingQueue and consists of elements that are of type Delayed. An element can be retrieved from this queue only after its delay period.
PriorityBlockingQueue	Equivalent to java.util.PriorityQueue, but implements the BlockingQueue interface.
SynchronousQueue	This class implements BlockingQueue. In this container, each insert() by a thread waits (blocks) for a corresponding remove() by another thread and vice versa.
LinkedBlockingDeque	This class implements BlockingDeque where insert and remove operations could block; uses a linked-list for implementation.
ConcurrentHashMap	Analogous to Hashtable, but with safe concurrent access and updates.
ConcurrentSkipListMap	Analogous to TreeMap, but provides safe concurrent access and updates.
ConcurrentSkipListSet	Analogous to TreeSet, but provides safe concurrent access and updates.
CopyOnWriteArrayList	Similar to ArrayList, but provides safe concurrent access. When the container is modified, it creates a fresh copy of the underlying array.
CopyOnWriteArraySet	A Set implementation, but provides safe concurrent access and is implemented using CopyOnWriteArrayList. When the container is modified, it creates a fresh copy of the underlying array.

M. Romdl

11

11



1Z0-809 Chapter 11

CopyOnWriteArrayList Class

- **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**.

M. Romdhani, 2020

12

12

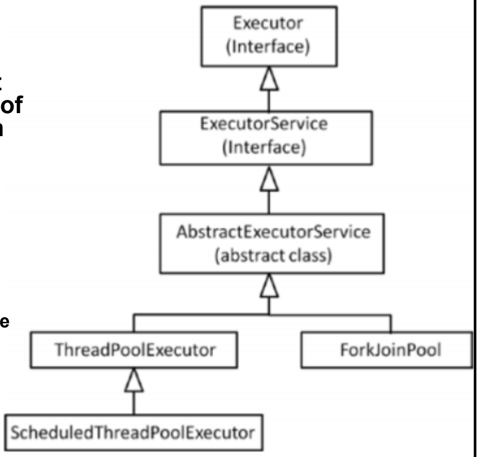
Use Executors and ThreadPools

13

1Z0-809 Chapter 11

The Exceutor interface

- You can directly create and manage threads in the application by creating Thread objects. However, if you want to abstract away the low-level details of multi-threaded programming, you can make use of the Executor interface.
- You can directly create and manage threads in the application by creating Thread objects. However, if you want to abstract away the low-level details of multi-threaded programming, you can make use of the Executor interface.
- Executor
  - Executor is an interface that declares only one method: void execute(Runnable).
  - This may not look like a significant interface by itself, but its derived classes (or interfaces), such as ExecutorService, ThreadPoolExecutor, and ForkJoinPool, support useful functionality.


$$\begin{tikzcd}
 & \text{Executor (Interface)} & \\
 & \text{ExecutorService (Interface)} \arrow[u] & \\
 & \text{AbstractExecutorService (abstract class)} \arrow[u] & \\
\text{ThreadPoolExecutor} \arrow[ur] & & \text{ForkJoinPool} \arrow[ul] \\
\text{ScheduledThreadPoolExecutor} \arrow[u] & &
\end{tikzcd}$$

M. Romdhani, 2020

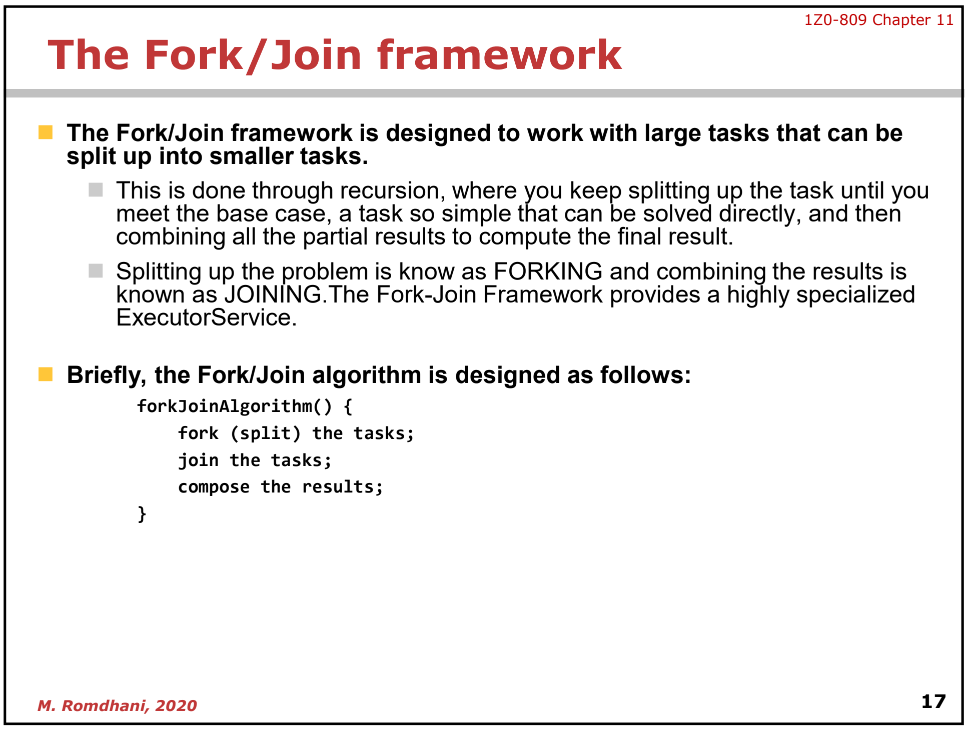
14

14

Callable and ExecutorService

- **Callable**
  - Callable is an interface that declares only one method: call(). Its full signature is V call() throws Exception. It represents a task that needs to be completed by a thread. Once the task completes, it returns a value
  - To execute a task using the Callable object, you first create a **thread pool**.
  - A thread pool is a collection of threads that can execute tasks.
- **ExecutorService**
  - The ExecutorService interface extends the Executor interface and provides services such as termination of threads and production of **Future** objects. Some tasks may take considerable execution time to complete.
- **Future**
  - Future represents objects that contain a value that is returned by a thread in the future (i.e., it returns the value once the thread terminates in the "future").
    - You can use the isDone() method in the Future class to check if the task is complete and then use the get() method to fetch the task result.
    - If you call the get() method directly while the task is not complete, the method blocks until it completes and returns the value

Use the Parallel Fork/Join Framework



1Z0-809 Chapter 11

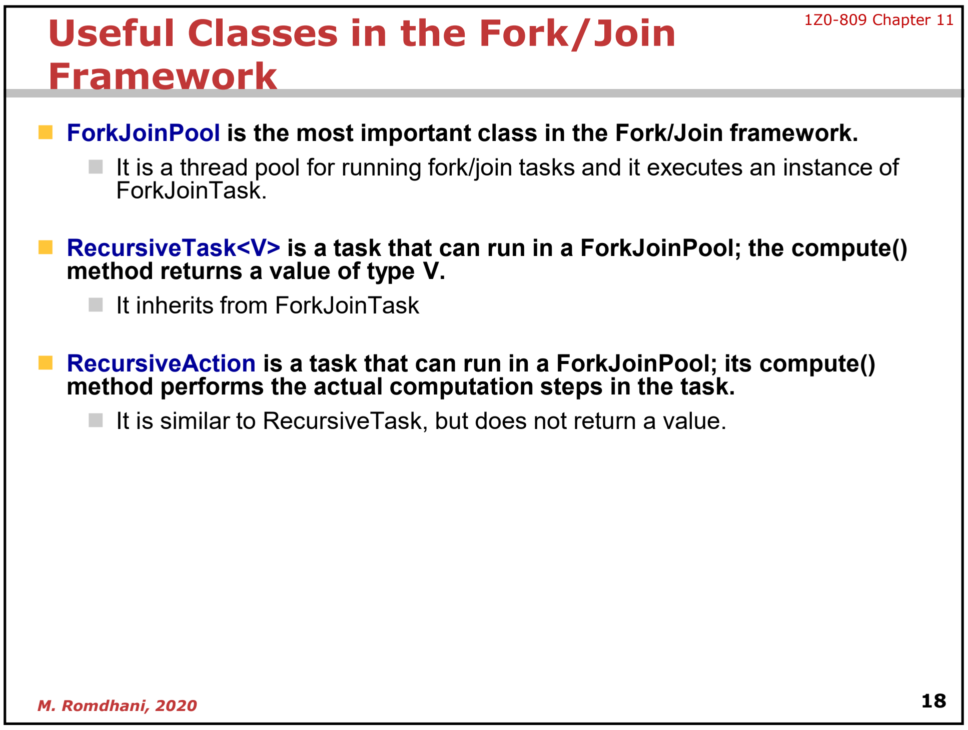
The Fork/Join framework

- **The Fork/Join framework is designed to work with large tasks that can be split up into smaller tasks.**
  - This is done through recursion, where you keep splitting up the task until you meet the base case, a task so simple that can be solved directly, and then combining all the partial results to compute the final result.
  - Splitting up the problem is know as FORKING and combining the results is known as JOINING.The Fork-Join Framework provides a highly specialized ExecutorService.
- **Briefly, the Fork/Join algorithm is designed as follows:**

```
forkJoinAlgorithm() {
    fork (split) the tasks;
    join the tasks;
    compose the results;
}
```

M. Romdhani, 2020 **17**

17



1Z0-809 Chapter 11

Useful Classes in the Fork/Join Framework

- **ForkJoinPool is the most important class in the Fork/Join framework.**
  - It is a thread pool for running fork/join tasks and it executes an instance of ForkJoinTask.
- **RecursiveTask<V> is a task that can run in a ForkJoinPool; the compute() method returns a value of type V.**
  - It inherits from ForkJoinTask
- **RecursiveAction is a task that can run in a ForkJoinPool; its compute() method performs the actual computation steps in the task.**
  - It is similar to RecursiveTask, but does not return a value.

M. Romdhani, 2020 **18**

18

Parallel Streams

19

How to Make a Parallel Stream Pipeline ?

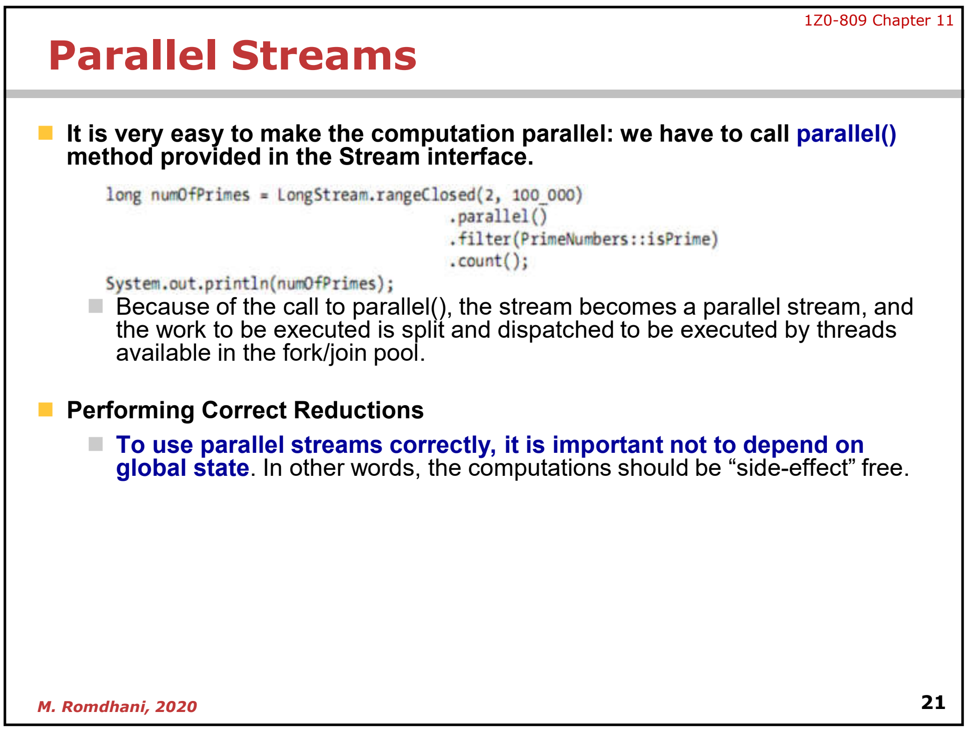
- **Parallel streams split the elements into multiple chunks, process each chunk with different threads, and (if necessary) combine the results from those threads to evaluate the final result.**
  - We discussed the fork/join framework: tasks are executed by recursively splitting them into sub-tasks and then the sub-tasks are executed in parallel. Parallel streams internally use this fork/join framework. The process steps should consist of stateless and independent tasks.
- **We create 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);
```

  - Let's take a peek at how the tasks created by this parallel stream get split up and handled by workers in the ForkJoinPool.

```
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);
```

20



Parallel Streams

- **It is very easy to make the computation parallel: we have to call parallel() method provided in the Stream interface.**

```
long numOfPrimes = LongStream.rangeClosed(2, 100_000)
                                            .parallel()
                                            .filter(PrimeNumbers::isPrime)
                                            .count();
System.out.println(numOfPrimes);
```

  - Because of the call to parallel(), the stream becomes a parallel stream, and the work to be executed is split and dispatched to be executed by threads available in the fork/join pool.

- **Performing Correct Reductions**
  - **To use parallel streams correctly, it is important not to depend on global state**. In other words, the computations should be "side-effect" free.