

11

Concurrency

Exam objectives covered in this chapter	What you need to know
[11.1] Use collections from the <code>java.util.concurrent</code> package with a focus on the advantages over and differences from the traditional <code>java.util</code> collections	How collections from the <code>java.util.concurrent</code> package resolve common concurrency issues How to replace traditional collections with concurrent collections
[11.2] Use <code>Lock</code> , <code>ReadWriteLock</code> , and <code>ReentrantLock</code> in the <code>java.util.concurrent.locks</code> package to support lock-free, thread-safe programming on single variables	How to exercise fine control over locking objects by using <code>Lock</code> , <code>ReadWriteLock</code> , and <code>ReentrantLock</code>
[11.3] Use <code>Executor</code> , <code>ExecutorService</code> , <code>Executors</code> , <code>Callable</code> , and <code>Future</code> to execute tasks using thread pools	How to separate task and threads using <code>Executor</code> How to manage a pool of threads by using <code>ExecutorService</code> How to define tasks using <code>Runnable</code> and <code>Callable</code> How to use <code>Executors</code> to access objects of <code>Executor</code> , <code>ExecutorService</code> , and thread pools
[11.4] Use the parallel fork/join framework	How to work with a fork/join framework using subclasses of <code>ForkJoinTask</code> and <code>ForkJoinPool</code>

With the increasing processing power of devices, your applications must support concurrent execution of tasks for faster outputs. Concurrent applications also make optimal use of the processors. But concurrent applications are difficult to develop, maintain, and debug. To develop thread-safe, high-performance, and scalable applications, Java's low-level threading capabilities are insufficient. This chapter outlines the common issues with the concurrent execution of tasks and how to combat these issues using various approaches, frameworks, classes, and interfaces. Again, the coverage of concurrency issues and their solutions is limited to the topics on the exam.

This chapter covers

- Using collections from the `java.util.concurrent` package
- Applying fine-grained locking using locking classes from the `java.util.concurrent.locks` package
- Using `ExecutorService` and pools of threads to execute and manage tasks
- Using the parallel fork/join framework

Let's get started with effectively resolving common concurrency problems by using collection classes from the `java.util.concurrent` package.

11.1 Concurrent collection classes



[11.1] Use collections from the `java.util.concurrent` package with a focus on the advantages over and differences from the traditional `java.util` collections

The concurrent collection classes avoid memory inconsistency errors by defining a happens-before relationship between an operation that adds an object to the collection and subsequent operations that access or remove that object. Developers have long been developing thread-safe versions of the collection objects from the `java.util` package. The `java.util.concurrent` package includes a number of additions to the Java Collections Framework. These are most easily categorized by the collection interfaces provided:

- `BlockingQueue` defines a first-in-first-out data structure that blocks or times out when you attempt to add items to a full queue, or retrieve from an empty queue.
- `ConcurrentMap` is a subinterface of `java.util.Map` that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of `ConcurrentMap` is `ConcurrentHashMap`, which is a concurrent analog of `HashMap`.
- `ConcurrentNavigableMap` is a subinterface of `ConcurrentMap` that supports approximate matches. The standard general-purpose implementation of

`ConcurrentNavigableMap` is `ConcurrentSkipListMap`, which is a concurrent analog of `TreeMap`.

Writing concurrent programs is difficult—you need to deal with thread safety and performance. The individual operations of `ConcurrentHashMap` are safe—that is, multiple threads can put values into the same map object in a safe manner. But these can be misused by the developers if they try to combine multiple safe operations into a single operation.

11.1.1 Interface BlockingQueue

The `BlockingQueue` interface is a queue that's safe to use when shared between multiple threads. The implementing classes like `ArrayBlockingQueue` include a constructor to define an initial capacity (which can't be modified) from which items are added and removed. It blocks adding new elements if the queue has reached its capacity. It also blocks removing elements from an empty queue. It works on the *producer-consumer* pattern, which is when a single thread or multiple threads produce elements and add them to a queue to be consumed by other threads.

Imagine multiple clients (producers) that send requests to a server. The server (consumer) responds to all the requests that it receives. To manage the requests that all the clients might send to the server, the server can limit the maximum number of requests that it can accept at a given point in time. The requests can be added to a blocking queue, which will block adding new requests if it reaches its upper limit. Similarly, if no new requests are available in a queue, the server thread will block until requests are made available to it. Here's an implementation of this example in code:

```
class Request {}

class Client implements Runnable {
    private BlockingQueue<Request> queue;
    Client(BlockingQueue<Request> queue) {
        this.queue = queue;
    }
    public void run() {
        try {
            Request req = null;
            while(true) {
                req = new Request();
                queue.put(req);
                System.out.println("added request - " + req);
            }
        }
        catch (InterruptedException ex) {
            System.out.println(ex);
        }
    }
}

class Server implements Runnable {
    private BlockingQueue<Request> queue;
    
```

Inserts Request objects into BlockingQueue, waiting if necessary for space to become available.

```

    Server(BlockingQueue<Request> queue) {
        this.queue = queue;
    }
    public void run() {
        try {
            while (true) {
                System.out.println("processing .. " + queue.take());
            }
        }
        catch (InterruptedException ex) {
            System.out.println(ex);
        }
    }
}

class LoadTesting{
    public static void main(String args[]) {
        BlockingQueue<Request> queue = new ArrayBlockingQueue<Request>(3);

        Client client = new Client(queue);
        Server server = new Server(queue);

        new Thread(client).start();
        new Thread(server).start();
    }
}

```

Retrieves and removes the head of BlockingQueue, waiting if necessary until a Request object becomes available.

Pass the same BlockingQueue object to client and server

client adds Request objects to queue and server retrieves them from the queue.

11.1.2 Interface ConcurrentHashMap

The ConcurrentHashMap interface extends the java.util.Map interface. It defines methods to replace or remove a key-value pair if the key is present, or add a value if the key is absent. Table 11.1 lists methods of ConcurrentHashMap.

Table 11.1 Methods of interface ConcurrentHashMap

Method	Description
V putIfAbsent(K key, V value)	If the specified key isn't already associated with a value, this associates it with the given value.
boolean remove(Object key, Object value)	Removes the entry for a key only if it's currently mapped to a given value.
V replace(K key, V value)	Replaces the entry for a key only if it's currently mapped to some value.
boolean replace(K key, V oldValue, V newValue)	Replaces the entry for a key only if it's currently mapped to a given value.

11.1.3 Class ConcurrentHashMap

A concrete implementation of the ConcurrentHashMap interface, class ConcurrentHashMap is a concurrent class analogous to class HashMap. A HashMap is an unsynchronized collection. If you're manipulating a HashMap using multiple threads, you must synchronize

its access. But locking the entire `HashMap` object can create serious performance issues when it's being accessed by multiple threads. If multiple threads are retrieving values, it makes sense to allow concurrent read operations and monitor write operations.

The ConcurrentHashMap class is the answer to improving the responsiveness of HashMap when it needs to be accessed concurrently by multiple threads. Instead of exclusively locking itself to be accessed by one thread, ConcurrentHashMap allows access by multiple threads. It concurrently allows multiple threads to read its values and limited threads to modify its values. Also, the iterators of ConcurrentHashMap don't throw a ConcurrentModificationException, so you don't need to lock the collection while iterating it. So what happens if new elements are added to ConcurrentHashMap *after* you accessed its iterator? The iterator may still traverse only the elements that existed at the time of creation of the iterator. Though not guaranteed on all platforms, the iterators might reflect the new additions.

With the added benefits of offering better performance when accessed by multiple threads concurrently, this collection also offers some drawbacks. Because it doesn't lock the complete collections while modifying their elements, methods like `size()` might not return the exact accurate size of a `ConcurrentHashMap` when invoked by multiple threads.

Let's work with an example of class `ConcurrentHashMap`:

```
class UseConcurrentMap {  
    static final ConcurrentHashMap<Integer, String> map =  
        new ConcurrentHashMap<>();  
  
    static {  
        //code to populate map  
    }  
  
    static void manipulateMap(Integer key, String value) {  
        // complex computations  
        if(!map.containsKey(key))  
            map.put(key, value);  
        else  
            map.replace(key, value);  
    }  
}
```

When you work with multiple threads, you need to synchronize access to your shared resources so that concurrent access doesn't leave them in an inconsistent state. In the preceding example, the individual operation `containsKey(key)` is a read operation and `put(key, value)` is a write operation. Though individually these methods are thread safe, together (execute method 1, then method 2) they aren't. A race condition can occur when method `manipulateMap()` is executed by multiple threads. The solution is to replace them with a single atomic method call:

```
static void manipulateMap(Integer key, String value) {  
    // complex computations  
    map.replace(key, value);  
}
```

← **Atomic operation replaces value in map if corresponding key is present.**

Before moving on to the next section, let's revisit the mapping of classes and interfaces from `java.util.concurrent` and its corresponding `java.util` analog, as shown in table 11.2.

Table 11.2 Mapping of classes and interfaces from package `java.util.concurrent` and its corresponding package `java.util` analog

Package <code>java.util.concurrent</code>	<code>java.util</code> analog
<code>BlockingQueue</code>	<code>Queue</code>
<code>ArrayBlockingQueue</code>	<code>Queue</code>
<code>LinkedBlockingQueue</code>	<code>Queue</code>
<code>ConcurrentMap</code>	<code>Map</code>
<code>ConcurrentHashMap</code>	<code>HashMap</code>
<code>ConcurrentSkipListMap</code>	<code>TreeMap</code>
<code>CopyOnWriteArrayList</code>	<code>ArrayList</code>
<code>LinkedBlockingDeque</code>	<code>Deque</code>

In chapter 10, we worked with synchronized methods and code blocks. Before a thread can execute a synchronized method's code it implicitly acquires a lock on an object's monitor. But these implicit locking techniques are inefficient to develop scalable concurrent applications. In the next section, we'll work with explicit lock objects that offer finer locking control.

11.2 Locks



[11.2] Use `Lock`, `ReadWriteLock`, and `ReentrantLock` in the `java.util.concurrent.locks` package to support lock-free, thread-safe programming on single variables

Lock objects offer multiple advantages over implicit locking of an object's monitor. Unlike an implicit lock, a thread can use explicit lock objects to wait to acquire a lock until a time duration elapses. Lock objects also support interruptible lock waits, non-block-structured locks, multiple condition variables, lock polling, and scalability benefits.



NOTE To execute synchronized code, a thread must acquire either an *implicit* or an *explicit* lock on an object's monitor. Where no explicit Lock classes are used, I'll refer to it as an implicit lock.

Table 11.3 shows a list of the methods of the `Lock` interface from the Java API documentation.

Table 11.3 Methods of interface Lock

Method	Description
void lock()	Acquires the lock. If the lock isn't available then the current thread becomes disabled for thread scheduling purposes and lies dormant until the lock has been acquired.
void lockInterruptibly()	Acquires the lock unless the current thread is interrupted.
Condition newCondition()	Returns a new Condition instance that's bound to this Lock instance.
boolean tryLock()	Acquires the lock only if it's free at the time of invocation.
boolean tryLock(long time, TimeUnit unit)	Acquires the lock if it's free within the given waiting time and the current thread hasn't been interrupted.
void unlock()	Releases the lock.

Let's get started with using class `ReentrantLock` that implements the `Lock` interface.

11.2.1 Acquire lock

Method `lock()` acquires a lock on a `Lock` object. If the lock isn't available, it waits until the lock can be acquired. For instance

```
class Rainbow {
    Lock myLock = new ReentrantLock();           ← Lock object
    static List<String> colors = new ArrayList<>();
    public void addColor(String newColor) {
        myLock.lock();                         ← Call lock to acquire lock;
                                                wait if lock not available
        try {
            colors.add(newColor);
        }
        finally {
            myLock.unlock();                  ← Call unlock to
                                                release lock
        }
    }
}
```

Method `lock()` is comparable to intrinsic locks because it waits until a lock can be acquired on a `Lock` object.



EXAM TIP Call method `unlock()` on a `Lock` object to release its lock when you no longer need it.

Let's see how you can poll and check whether method `lock()` is available, without waiting for it, in the next section.

11.2.2 Acquire lock and return immediately

Imagine you're waiting for your favorite soccer star to sign an autograph for you. If you've been waiting for too long, it makes sense to quit waiting and leave. But threads waiting to acquire implicit object locks can't quit. Once a thread initiates a request to acquire an implicit lock on an object monitor, it can neither stop itself nor can it be asked to do so by any other thread. With explicit locks, you can request a thread to acquire a lock on an object monitor if it's available and return immediately.

Following is example code in which class `Order` uses implicit locks on two reference variables *before* it can manipulate them. When method `main()` starts two instances of class `Shipment`, passing them `Inventory` objects in reverse order, they might deadlock, without generating any output:

```
class Inventory {
    int inStock; String name;
    Inventory(String name) { this.name = name; }
    public void stockIn(long qty) { inStock += qty; }
    public void stockOut(long qty) { inStock -= qty; }
}
class Shipment extends Thread {
    Inventory loc1, loc2; int qty;
    Shipment(Inventory loc1, Inventory loc2, int qty) {
        this.loc1 = loc1;
        this.loc2 = loc2;
        this.qty = qty;
    }
    public void run() {
        synchronized(loc1) {
            synchronized(loc2) {
                loc2.stockOut(qty);
                loc1.stockIn(qty);
            }
            System.out.println(loc1.inStock + ":" + loc2.inStock);
        }
    }
    public static void main(String args[]) {
        Inventory loc1 = new Inventory("Seattle"); loc1.inStock = 100;
        Inventory loc2 = new Inventory("LA"); loc2.inStock = 200;
        Shipment s1 = new Shipment(loc1, loc2, 1);
        Shipment s2 = new Shipment(loc2, loc1, 10);
        s1.start();
        s2.start();
    }
}
```

Depending on how an underlying scheduler executes the threads `s1` and `s2`, they might or might not deadlock. Let's modify the preceding example and add an explicit lock object to class `Inventory` on single variables `loc1` and `loc2`, so that threads `s1` and `s2` *never* deadlock. Now, instead of locking on the `Inventory` object's monitor,

method `run()` in `Shipment` can lock on an explicit lock object, `ReentrantLock` (modifications in bold):

```

import java.util.concurrent.locks.*;
class Inventory {
    int inStock; String name;
    Lock lock = new ReentrantLock();
    Inventory(String name) { this.name = name; }
    public void stockIn(long qty) { inStock += qty; }
    public void stockOut(long qty) { inStock -= qty; }
}
class Shipment extends Thread {
    Inventory loc1, loc2; int qty;
    Shipment(Inventory loc1, Inventory loc2, int qty) {
        this.loc1 = loc1;
        this.loc2 = loc2;
        this.qty = qty;
    }
    public void run() {
        if (loc1.lock.tryLock()) {
            if (loc2.lock.tryLock()) {
                loc2.stockOut(qty);
                loc1.stockIn(qty);
                System.out.println(loc1.inStock + ":" + loc2.inStock);
                loc2.lock.unlock();
                loc1.lock.unlock();
            } else
                System.out.println("Locking false:" + loc2.name);
        } else
            System.out.println("Locking false:" + loc1.name);
    }
    public static void main(String args[]) {
        Inventory loc1 = new Inventory("Seattle"); loc1.inStock = 100;
        Inventory loc2 = new Inventory("LA"); loc2.inStock = 200;
        Shipment s1 = new Shipment(loc1, loc2, 1);
        Shipment s2 = new Shipment(loc2, loc1, 10);
        s1.start();
        s2.start();
    }
}

```

At ①, the code defines a reference variable of type `Lock`, which is assigned an object of class `ReentrantLock`. When started, the thread `Shipment` must acquire a lock on *both* `Inventory` objects `loc1` and `loc2` so that it can execute methods `stockOut()` and `stockIn()`, thereby ensuring that no other thread is modifying these objects. With explicit locks, `Shipment` can acquire a lock on object `lock`, which is defined as an instance member of class `Inventory`.

At ②, `run()` calls `tryLock()` on `loc1.lock`. Method `tryLock()` tries to acquire a lock on a `Lock` object, and returns immediately returning a boolean value specifying whether it could obtain the lock or not. If `loc1.lock.tryLock()` returns `true`,

`loc2.lock.tryLock()` at ❸ tries to obtain a lock on `loc2.lock` before it can manipulate `loc1` and `loc2` at ❹. The code outputs appropriate messages if it can't lock at ❺.

Method `main()` starts two new threads, `s1` and `s2`, passing them objects `loc1` and `loc2`. No matter how threads `s1` and `s2` execute, they will never deadlock. Unlike waiting to acquire an implicit lock on objects `loc1` and `loc2`, they call `loc1.lock.tryLock()` and `loc2.lock.tryLock()`, which return immediately.



EXAM TIP Watch out for the use of methods `acquire()`, `acquireLock()`, `release()`, and `releaseLock()` on the exam. None of these is a valid method. Because the terms *acquire* and *release* are used to discuss methods `lock()`, `unlock()`, `tryLock()`, and `lockInterruptibly()`, these terms might be used on the exam to confuse you.

11.2.3 Interruptible locks

Imagine you have an appointment with a doctor and are waiting for her to arrive. Your waiting can be interrupted by a phone call informing you that you need to attend to other tasks. Also, you might set yourself a time limit, after which you might not be able to wait and will resume your other work.

The following methods of `Lock` enable you to specify a waiting timeout or to try and acquire a lock while being available for interruption:

- `lockInterruptibly()`
- `tryLock(long time, TimeUnit unit)`

In listing 11.1, class `Bus` defines a `Lock` object. Class `Employee` extends class `Thread`. When started, it tries to acquire a lock on the `Lock` object associated with a `Bus` instance using `lockInterruptibly()`.

Listing 11.1 Working with interruptible locks

```
import java.util.concurrent.locks.*;
class Bus {
    Lock lock = new ReentrantLock();
    public void boardBus(String name) {
        System.out.println(name + ": boarded");
    }
}
class Employee extends Thread {
    Bus bus;
    String name;
    Employee(String name, Bus bus) {
        this.bus = bus;
        this.name = name;
    }
    public void run() {
        try {
            bus.lock.lockInterruptibly();
            try {
                bus.boardBus(name);
            }
            catch (InterruptedException e) {
                System.out.println("Thread " + name + " interrupted");
            }
        }
        catch (InterruptedException e) {
            System.out.println("Thread " + name + " interrupted");
        }
    }
}
```

❶ Try to acquire lock while being available for interruption

❷ If lock acquired, execute required code

```

        } finally {
            bus.lock.unlock();
        }
    } catch(InterruptedException e) {
        System.out.println(name + ": Interrupted!!!");
        Thread.currentThread().interrupt();
    }
}
public static void main(String args[]) {
    Bus bus = new Bus();
    Employee e1 = new Employee("Paul", bus);
    e1.start();
    e1.interrupt();
    Employee e2 = new Employee("Shreya", bus);
    e2.start();
}
}

```

The diagram illustrates the execution flow of the code. It shows six numbered steps: 1. Start thread e1 (at the beginning of the main() method); 2. Execute code (the try block); 3. Release acquired lock in finally block (the finally block releases the lock); 4. Define action if thread is interrupted (the catch block handles InterruptedException); 5. Start thread e1 (the Employee e1 object starts its run() method); 6. Interrupt thread e1 (the Employee e1 object's interrupt() method is called). Arrows indicate the flow from step 1 to 2, 2 to 3, 3 to 4, 4 to 5, and 5 to 6.

In the preceding code, the outer try-catch blocks at ① and ④ acquire the lock and handle the `InterruptedException`. If the locking attempt is successful, the try block at ② executes the required code and its finally block at ③ releases the lock. The code at ⑤ starts a thread and the code at ⑥ tries to interrupt the thread. Even though the code at ⑥ interrupts the thread `e1`, do you think it's sure to be interrupted? Not really, depending on how the thread `e1` is executed—that is, whether it's waiting to acquire a lock on `bus.lock` or it has already completed the code in method `run()`, the code at ⑥ might not interrupt `e1`. Also, because the thread `e2` might actually start its execution before the thread `e1`, figure 11.1 shows the probable outputs of the preceding example.

Shreya: boarded	Paul: Interrupted!!	Paul: boarded	Shreya: boarded
	Shreya: boarded	Shreya: boarded	Paul: boarded

Figure 11.1 Probable outputs of listing 11.1

It's important to make note of the order in which you acquire a lock using `lockInterruptibly()`, handle the `InterruptedException` that it throws, and unlock it. Let's see whether you can detect this point in the code (you might be tested on it on the exam!) using the next "Twist in the Tale" exercise.

Twist in the Tale 11.1

What is the probable output of the following code?

```

import java.util.concurrent.locks.*;
class Bus {
    Lock lock = new ReentrantLock();

```

```

        public void boardBus(String name) {
            System.out.println(name + ": boarded");
        }
    }
    class Employee extends Thread {
        Bus bus; String name;
        Employee(String name, Bus bus) {
            this.bus = bus;
            this.name = name;
        }
        public void run() {
            try {
                bus.lock.lockInterruptibly();
                bus.boardBus(name);
            }
            catch (InterruptedException e) {
                System.out.println(name + ": Interrupted!!!");
                Thread.currentThread().interrupt();
            }
            finally {
                bus.lock.unlock();
            }
        }
        public static void main(String args[]) {
            Employee e1 = new Employee("Paul", new Bus());
            e1.start();
            e1.interrupt();
        }
    }
}

```

- a** Paul: boarded
 - b** Paul: Interrupted!!
 - c** Paul: Interrupted!! and IllegalThreadStateException is thrown
 - d** Paul: Interrupted!! and IllegalMonitorStateException is thrown
 - e** None of the above
-

In the next section, you'll see how you can retain a lock on `Lock` objects, across methods.

11.2.4 Nonblock-structured locking

With intrinsic locks, you must release the lock on an object's monitor at the end of the synchronized code blocks or methods. Because code blocks can't span across methods, intrinsic locks can't be acquired across methods. Extrinsic locks or a lock on `Lock` objects can be acquired across methods. Figure 11.2 compares block and nonblock locking with intrinsic and extrinsic locks.

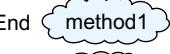
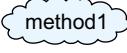
	Intrinsic locks	Extrinsic locks
Aquire lock	Start 	Start 
Release lock		End 
	Start synchronized method1 — — — — — — — — — — — — — — — — End synchronized 	Start synchronized block — — — — — — — — — — — — — — — — End synchronized block

Figure 11.2 Comparing intrinsic and extrinsic locks for block and nonblock locking

Here's some example code to work with extrinsic locks:

```
import java.util.concurrent.locks.*;
class Bus {
    ReentrantLock lock = new ReentrantLock();
    boolean locked = false;
    public void board(String name) {
        if (lock.tryLock()) { ←
            locked = true;
            System.out.println(name + ": boarded");
        }
    }
    public void deboard(String name) {
        if (lock.isHeldByCurrentThread() && locked) {
            System.out.println(name + ": deboarded");
            lock.unlock(); ←
            locked = false;
        }
    }
}
```

Acquire lock in one method.

Release lock in another method.



EXAM TIP Lock fairness: a `ReentrantLock` lock can be a nonfair or a fair lock. It can acquire locks in the order they were requested or acquire locks out of turn.

11.2.5 Interface `ReadWriteLock`

Interface `ReadWriteLock` maintains a pair of associated locks, one for read-only operations and another for write-only operations. The read-only lock may be held simultaneously by multiple reader threads as long as there are no writing processes in progress. The write-only lock is an exclusive lock. It can be acquired by only one thread. As listed in table 11.4, the `ReadWriteLock` interface defines only two methods: `readLock()` and `writeLock()`.

Table 11.4 Methods of interface `ReadWriteLock`

Method name	Description
<code>Lock readLock()</code>	Returns the lock used for reading.
<code>Lock writeLock()</code>	Returns the lock used for writing.



EXAM TIP The `ReadWriteLock` interface doesn't extend `Lock` or *any* other interface. It maintains a pair of associated `Locks`—one for only reading operations and one for writing.

You can use methods `readLock()` and `writeLock()` to get a reference to the read or write `Lock`. Let's work with a concrete implementation of the `ReadWriteLock` interface, class `ReentrantReadWriteLock`, in the next section.

11.2.6 Class `ReentrantReadWriteLock`

A `ReentrantReadWriteLock` has a read and a write lock associated with it. You can access these locks (reference variables of type `Lock`) by calling its methods `readLock()` and `writeLock()`. You can acquire multiple read locks as long as no write lock has been acquired on a `ReadWriteLock` object. The `writeLock` is an exclusive lock; it can be acquired by only one thread when no read thread has been acquired. For example

```
import java.util.*;
import java.util.concurrent.locks.*;
class Rainbow {
    private final ReadWriteLock myLock = new ReentrantReadWriteLock();
    private static int pos;
    static Map<Integer, String> colors = new HashMap<>();
    public void addColor(String newColor) {
        myLock.writeLock().lock(); ←
        try {
            colors.put(new Integer(++pos), newColor); ←
        }
        finally {
            myLock.writeLock().unlock(); ←
        }
    }
}
```

ReentrantReadWriteLock

**myLock.writeLock() returns
writeLock and lock acquires
a lock on it.**

**Release lock
on writeLock.**

```

public void display() {
    myLock.readLock().lock();           ←
    try {
        for (String s : colors.values()) {
            System.out.println(s);
        }
    } finally {
        myLock.readLock().unlock();      ←
    }
}

```

myLock.readLock()
returns **readLock** and
lock acquires a lock on it.

**Unlock
readLock.**

11.2.7 Atomic variables

The `java.util.concurrent.atomic` package defines multiple classes that support atomic operations of read-compare/modify-write on single variables. At the surface, these operations might seem similar to the operations with volatile variables. Though modifications to a volatile variable are guaranteed to be visible to other threads, volatile variables can't define a sequence of operations (like read-compare/modify-write) as an atomic operation.

Here's an example of class `Book` from chapter 10 to show you how nonatomic operations with primitive variables can lead to thread interference:

```

class Book{
    String title;
    int copiesSold = 0;
    Book(String title) {
        this.title = title;
    }
    public void newSale() {
        ++copiesSold;          ←
    }
    public void returnBook() {
        --copiesSold;         ←
    }
}

```

① **Nonatomic statements include loading
of variable values from memory to
registers, manipulating values, and
loading them back to memory.**

The code defined at ① isn't atomic. Incrementing or decrementing primitive values includes multiple steps. When executed by multiple concurrent threads, `newSale()` and `returnBook()` can result in thread interference. To get around this, you can define these methods as synchronized methods, but it will block thread execution. Java defines multiple convenient classes in the `java.util.concurrent.atomic` package that define frequently used operations like read-modify-write as atomic operations. Let's use one of these classes, `AtomicInteger`, in class `Book`, and replace the type of its primitive `int` variable `copiesSold`:

```

class Book{
    String title;
    AtomicInteger copiesSold = new AtomicInteger(0);
}

```

```

Book(String title) {
    this.title = title;
}
public void newSale() {
    copiesSold.incrementAndGet();
}
public void returnBook() {
    copiesSold.decrementAndGet();
}
}

```



Methods `incrementAndGet()` and `decrementAndGet()` defined at ① are atomic operations. Concurrent execution of these methods won't result in interfering threads.

Class `AtomicInteger` defines multiple methods `xxxAndGet()` and `getAndXxx()`, where `Xxx` refers to an operation like increment, decrement, and add. `xxxAndGet()` returns an updated value and `getAndXxx()` returns the previous value.



EXAM TIP Method `incrementAndGet()` returns the updated value but method `AtomicInteger`'s `getAndIncrement()` returns the previous value.

The other commonly used operations with `AtomicInteger` are to add or subtract specified values from it, assign it a value, and compare values before assignment. Table 11.5 lists these methods.

Table 11.5 Methods of class `AtomicInteger`

Method	Description
<code>int addAndGet(int delta)</code>	Atomically adds the given value to the current value. Returns the updated value.
<code>int getAndAdd(int delta)</code>	Atomically adds the given value to the current value.
<code>compareAndSet(int expect, int update)</code>	Atomically sets the value to the given updated value if the current value == the expected value.
<code>int getAndSet(int newValue)</code>	Atomically sets to the given value and returns the old value.
<code>public final void set(int newValue)</code>	Sets to the given value.
<code>int getAndDecrement()</code>	Atomically decrements by one the current value. Returns the previous value.
<code>int getAndIncrement()</code>	Atomically increments by one the current value. Returns the previous value.
<code>int decrementAndGet()</code>	Atomically decrements by one the current value. Returns the updated value.
<code>int incrementAndGet()</code>	Atomically increments by one the current value. Returns the updated value.



EXAM TIP Class `AtomicInteger` defines method `compareAndSet()` but not method `setAndCompare()`.

Other commonly used classes defined in the `java.util.concurrent.atomic` package are `AtomicBoolean`, `AtomicLong`, `AtomicIntegerArray`, `AtomicLongArray`, and `AtomicReference<V>`. `AtomicLong` defines the same methods as class `AtomicInteger` (as listed in table 11.5). The difference is the type of method parameters and their return types (`long` instead of `int`).



EXAM TIP The `java.util.concurrent.atomic` package doesn't define classes by the names `AtomicShort`, `AtomicByte`, `AtomicFloat`, or `AtomicDouble`. These invalid class names might be used on the exam.

In this section, you learned how to exercise finer control over how locks are acquired, managed, and released. In the next section, we'll work with exercising finer control over the tasks done by threads and the threads themselves.

11.3 Executors



[11.3] Use `Executor`, `ExecutorService`, `Executors`, `Callable`, and `Future` to execute tasks using thread pools

In chapter 10, we used class `Thread` and the `Runnable` interface to create multiple threads that execute asynchronously. Class `Thread` and the `Runnable` interface define a close connection with a *task* (that is, a logical unit of work) done by a thread and the thread itself. Though okay for smaller applications, large applications call for a separation of tasks and the threads for thread creation and their management.

The `Executor` framework enables decoupling of task submission with task execution. By using this framework, you can create tasks using interfaces `Runnable` and `Callable`. These tasks are submitted to `Executor` to launch new tasks. `ExecutorService` extends `Executor` and adds methods to manage the lifecycle of tasks and executors. `ScheduledExecutorService` extends `ExecutorService` and supports future or periodic execution of tasks. `Future` represents the state of asynchronous tasks and can be used to query their status or cancel them. Class `Executors` defines utility and factory methods for interfaces `Executor`, `ExecutorService`, and `ScheduledExecutorService`.

Figure 11.3 shows the classes and interfaces that we'll work with in this section, how they're related to each other, and where they're placed in the `java.util.concurrent` package.

Let's get started with the `Executor` interface.

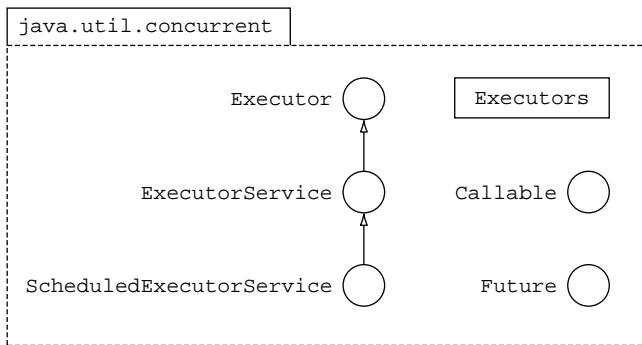


Figure 11.3 Interfaces and classes that we'll work with in this section

11.3.1 Interface Executor

With class Thread and the Runnable interface, the task executed by a thread and the thread itself are closely connected. The Executor interface allows you to define classes that know *how* to execute Runnable tasks. It allows you to decouple task submission and its execution. By implementing its sole method, void execute(Runnable), you can determine how you want to execute the tasks:

- Which task will execute first
- The order of execution of tasks
- How many tasks can execute concurrently
- How many tasks can be queued

Here's an example of a Runnable object, Order, which is processed by Hotel, an Executor:

```

class Order implements Runnable {
    String name;
    Order(String name) {this.name = name;}
    public void run() {
        System.out.println(name);
    }
}
class Hotel implements Executor {
    final Queue<Runnable> custQueue = new ArrayDeque<>();
    public void execute(Runnable r) {
        synchronized(custQueue) {
            custQueue.offer(r);           ① Implement execute.
        }
        processEarliestOrder();        ② Add Runnable object to a queue.
    }
    private void processEarliestOrder() {
        synchronized(custQueue) {
            Runnable task = custQueue.poll(); ③ Retrieve Runnable object from execute.
            new Thread(task).start();        ④ Start new thread for executing submitted task
        }
    }
}
  
```

In the preceding code, class Hotel controls how it processes the Runnable tasks submitted to it. At ①, Hotel implements Executor's method execute(). At ②, it adds the submitted task to a queue. Method execute() calls processEarliestOrder(), which at ③ retrieves a task from the queue, and at ④ starts a new thread to execute it.

By decoupling task submission from task execution, it's simple to modify how to execute tasks. In the preceding code, you can modify execute() so that class Hotel works with a pool of worker threads ready to be assigned a task to execute, rather than executing each task using a new thread.

You can code execute() to exercise complete control over how to execute tasks. You might wish to process tasks with high priority earlier than the lower-priority ones, or process the last submitted task as the first task.

In the next “Twist in the Tale” exercise, let's modify just a line of code from the preceding example and see whether you can determine its overall impact on the execution of the tasks submitted to class Hotel.

Twist in the Tale 11.2

What is the output of the following code?

```
class Order implements Runnable {  
    String name;  
    Order(String name) {this.name = name;}  
    public void run() {  
        System.out.println(name);  
    }  
}  
class Hotel implements Executor {  
    final Queue<Runnable> custQueue = new ArrayDeque<>();  
    public void execute(Runnable r) {  
        synchronized(custQueue) {  
            custQueue.offer(r);  
        }  
        processEarliestOrder();  
    }  
  
    private void processEarliestOrder() {  
        synchronized(custQueue) {  
            Runnable task = custQueue.poll();  
            task.run();  
        }  
    }  
    public static void main(String args[]) {  
        Hotel hotel = new Hotel();  
        hotel.execute(new Order("tea"));  
        hotel.execute(new Order("coffee"));  
        hotel.execute(new Order("burger"));  
    }  
}
```

- a The code will fail to compile.
- b The code will throw an exception at runtime.

- c The code can output "tea", "coffee", and "burger" in any order.
 - d The code will output "tea", "coffee", and "burger" in a fixed order for all executions of class Hotel.
-

The `java.util.concurrent` package defines an advanced interface, `ExecutorService`, which enables you to control and manage the submitted tasks better. Apart from accepting `Runnable` objects, it also accepts `Callable` objects. Before exploring the `ExecutorService` interface, let's examine the `Callable` interface.

11.3.2 Interface Callable

Comparing interfaces `Runnable` and `Callable`, method `run()` of the `Runnable` interface doesn't return a value and can't throw a checked exception. Both of these are taken care of by the `Callable` interface:

```
public interface Callable<V> {
    /**
     * Computes a result, or throws an exception if unable to do so.
     *
     * @return computed result
     * @throws Exception if unable to compute a result
     */
    V call() throws Exception;
}
```

In the following example, class `Order` implements the `Callable` interface. Because it isn't interested in returning a value from `call()`, it uses `Void` as its parameterized argument:

```
class Order implements Callable<Void> {
    String name;
    Order(String name) {this.name = name;}
    @Override
    public Void call() throws Exception {
        System.out.println(name);
        if (name.equalsIgnoreCase("berry"))
            throw new Exception("Berry unavailable");
        return null;
    }
}
```



EXAM TIP If you don't want your `Callable` to return a value, you can create it using `Callable<Void>`.

You can submit `Callable` and `Runnable` objects to an `ExecutorService`. Let's move forward with discussing the `ExecutorService` interface in the next section.

11.3.3 Interface ExecutorService

The ExecutorService interface extends the Executor interface and defines methods to manage progress and termination of tasks that are submitted to it. It defines methods to

- Submit single Runnable and Callable objects for execution, returning Future objects
- Submit multiple Runnable objects for execution, returning Future objects
- Shut down the ExecutorService, allowing or disallowing submitted tasks to be completed

Table 11.6 shows the methods of the ScheduledService interface.

Table 11.6 Methods of interface ScheduledService

Method name	Description
<code>boolean awaitTermination(long timeout, TimeUnit unit)</code>	Blocks until all tasks have completed execution after a shutdown request, the timeout occurs, or the current thread is interrupted—whichever happens first.
<code><T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)</code>	Executes the given tasks, returning a list of Futures holding their statuses and results when all complete.
<code><T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)</code>	Executes the given tasks, returning a list of Futures holding their statuses and results when all complete or the timeout expires—whichever happens first.
<code><T> T invokeAny(Collection<? extends Callable<T>> tasks)</code>	Executes the given tasks, returning the result of one that has completed successfully (that is, without throwing an exception), if any do.
<code><T> T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)</code>	Executes the given tasks, returning the result of one that has completed successfully (that is, without throwing an exception), if any do before the given timeout elapses.
<code>boolean isShutdown()</code>	Returns true if this executor has been shut down.
<code>boolean isTerminated()</code>	Returns true if all tasks have completed following shutdown.
<code>void shutdown()</code>	Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.
<code>List<Runnable> shutdownNow()</code>	Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.

Table 11.6 Methods of interface ScheduledService (continued)

Method name	Description
<T> Future<T> submit(Callable<T> task)	Submits a value-returning task for execution and returns a Future representing the pending results of the task.
Future<?> submit(Runnable task)	Submits a Runnable task for execution and returns a Future representing that task.
<T> Future<T> submit(Runnable task, T result)	Submits a Runnable task for execution and returns a Future representing that task.

In the next section, we'll work with thread pools, and you'll learn what they are, why you need them, and how they're implemented using ExecutorService.

11.3.4 Thread pools

Imagine only one chef is employed in a restaurant to prepare orders from its customers. As the restaurant consistently increases its customer count, it makes sense to employ more chefs to speed up service and reduce waiting time. But an increased count of chefs wouldn't decrease the food preparation time in the same proportion, because they would be waiting to access the same resources—that is, the oven, refrigerator, food-processing equipment, etc. Also, even if they aren't doing anything, they would consume restaurant resources like monetary benefits, physical space in the kitchen, and more. What happens when the kitchen runs out of all its physical space from the constant addition of new chefs? In an application, a similar condition called *exhaustion of physical resources* can lead to a crash.

By limiting the number of concurrent tasks, an application ensures that it doesn't fail or have resource exhaustion or suffer performance issues. The solution is to use pools of threads, which create a predefined number of threads and reuse them to execute tasks. You must optimize the size of the thread pool. The threads shouldn't overwhelm the scheduler and introduce unnecessary thread contention and performance degradation. There should be enough to only keep the processor busy.

A thread pool includes a homogeneous pool of worker threads. These are usually bound to a work queue, holding references to tasks that are waiting to be executed. A worker thread would typically request the next task for execution, execute it, and go back to waiting to execute the next task.

Class Executors in the `java.util.concurrent` package defines convenient static methods to retrieve multiple preconfigured thread pools:

- Fixed thread pool, which creates a pool with a fixed number of threads
- Cached thread pool
- Single thread executor
- Scheduled thread pool

Let's work with an example of Callable objects that are submitted to a thread pool:

```

class Order implements Callable<Void> {
    String name;
    Order(String name) {this.name = name;}
    public Void call() throws Exception { ← call() doesn't return any
        System.out.println(name);
        if (name.equalsIgnoreCase("berry"))
            throw new Exception("Berry unavailable");
        return null;
    }
}
class Hotel {
    ExecutorService service = Executors.newFixedThreadPool(5); ← Create ExecutorService
    object by calling
    Executors.newFixed-
    ThreadPool.

    public void orderFood(Order order) {
        service.submit(order); ← Submit Callable to
    }                               ExecutorService for execution.

    public void hotelClosedForDay() {
        service.shutdown(); ← Disable new tasks
    }                               from being submitted.

    public void hotelClosedForever() {
        service.shutdown(); ← Cancel
        try {                     currently
            if (!service.awaitTermination(60, TimeUnit.SECONDS)) { executing
                service.shutdownNow(); tasks.
            }
        } if (!service.awaitTermination(60, TimeUnit.SECONDS)) ←
            System.err.println("Pool did not terminate"); ← (Re-)cancel if current
        } catch (InterruptedException ie) { thread also interrupted
            service.shutdownNow();
            Thread.currentThread().interrupt();
        }
    }
}

Wait awhile for existing tasks to terminate. ← Wait awhile for tasks to respond to being cancelled.

```

In the preceding example, class Hotel uses Executors to retrieve a fixed thread pool of five threads. This thread pool implements the ExecutorService interface. When a task is submitted to ExecutorService using execute(), it uses one of its available worker threads to execute it. If no worker threads are available to execute the task, the submitted task waits for a worker thread to become available. Method shutdown() doesn't accept submission of new tasks and waits for the existing tasks to complete to execution. Method shutdownNow() cancels currently executing tasks, apart from refusing to accept new tasks. Method awaitTermination() blocks until all tasks have completed execution after a shutdown request, the timeout occurs, or the current thread is interrupted—whichever happens first.

11.3.5 Interface ScheduledExecutorService

The ScheduledExecutorService interface supports future or periodic execution of tasks.

Imagine you need to send out reminder emails to all the employees of your organization to submit their daily status reports. This email is sent out every day. Let's see how you can use ScheduledExecutorService in this case:

```

import static java.util.concurrent.TimeUnit.*;
import java.util.concurrent.*;
class Reminder implements Runnable {
    public void run() {
        // send reminder emails to all employees
        System.out.println("All Mails sent");
    }
}
class ReminderMgr {
    ScheduledExecutorService service =
        Executors.newScheduledThreadPool(1);
    Reminder reminder = new Reminder();

    public void sendReminders() {
        service.scheduleAtFixedRate(reminder, 0, 24, HOURS);
    }
    public static void main(String args[]) {
        ReminderMgr mgr = new ReminderMgr();
        mgr.sendReminders();
    }
}

```

Table 11.7 lists all the methods of the ScheduledExecutorService interface.

Table 11.7 Methods of interface ScheduledExecutorService

Method name	Description
<V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit)	Creates a ScheduledFuture that becomes enabled after the given delay.
ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)	Creates and executes a one-shot action that becomes enabled after the given delay.
ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)	Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently within the given period—that is, executions will commence after initialDelay, then initialDelay + period, then initialDelay + 2 × period, and so on.
ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)	Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently within the given delay between the termination of one execution and the commencement of the next.

The `ScheduledExecutorService` interface greatly simplifies thread handling for developers. It provides methods to manage tracking of progress and termination of asynchronous tasks (threads). In their absence, developers used to write their own classes or work with third-party classes to manage concurrent applications. Starting with version 7, Java has a new addition to the `java.concurrent` package, the fork/join framework. As you'll see in the next section, it makes the best use of multicore processors.

11.4 Parallel fork/join framework



[11.4] Use the parallel fork/join framework

Today, all computing devices—servers, desktops, tables, and mobile phones—feature multicore processors, so it made a lot of sense to add a feature to the programming language itself to utilize them. Introduced with Java 7, the fork/join framework extends the existing Java concurrency package, supporting hardware parallelism, a key feature of multicore systems. The fork/join framework isn't intended to replace or compete with the existing concurrency classes from the `java.util.concurrent` package. It works by breaking down larger, processing-intensive tasks into smaller, independent tasks recursively, processing each unit of a task and then merging back the results.



EXAM TIP The fork/join framework is best suited for tasks that are processor intensive and that can be divided into smaller tasks that can execute independently, in parallel. Tasks that block, work with I/O, or need synchronization aren't good candidates to use with the fork/join framework.

Figure 11.4 shows the logic of the fork/join framework's divide-and-conquer parallel algorithm.

As you can see in figure 11.4, the size of a problem is evaluated *before* it's further divided. If the problem is small enough (smaller than or equal to a threshold limit), it isn't subdivided and is executed sequentially. The bigger problems are subdivided into two or more subproblems, and the fork/join framework recursively invokes itself on the subproblems in parallel, waits for their results, and then combines them. You should select the threshold limit carefully.



NOTE The fork/join framework is named so because it initiates execution of a task that *forks* or starts multiple subtasks, and waits for them to *join* back (or complete their execution).

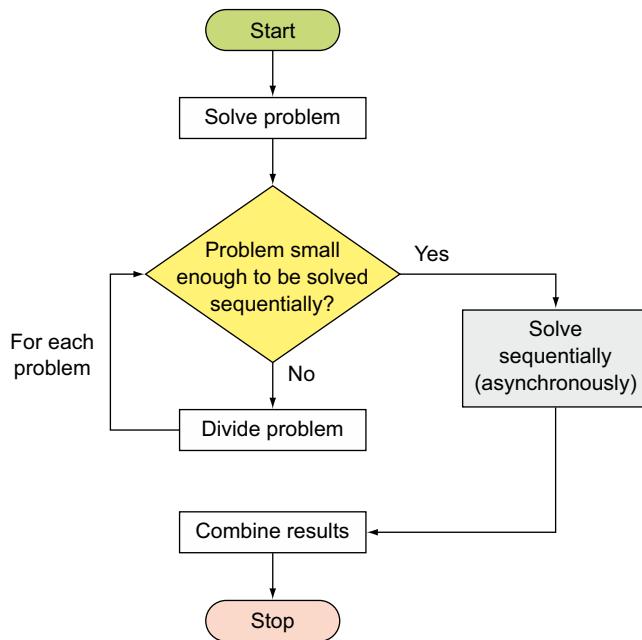


Figure 11.4 Divide-and-conquer algorithm used by fork/join framework

Class `ForkJoinPool` is a concrete implementation of the fork/join framework. It implements the `ExecutorService` interface. Like `ExecutorService`, a fork/join framework maintains a queue of tasks that are used to assign tasks to its multiple worker threads. But it's different from an `ExecutorService` because a fork/join framework implements

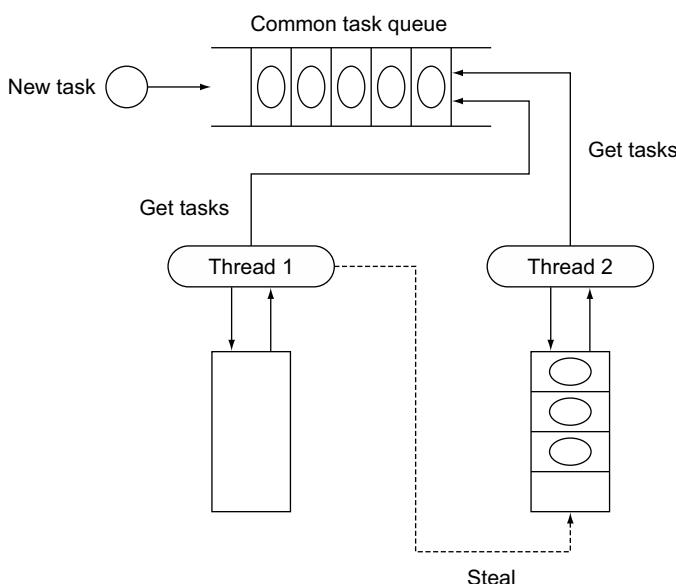


Figure 11.5 Threads in a fork/join framework use work-stealing algorithms. When a thread runs out of tasks in its own deque, it can steal tasks from other threads to avoid blocking waiting threads.

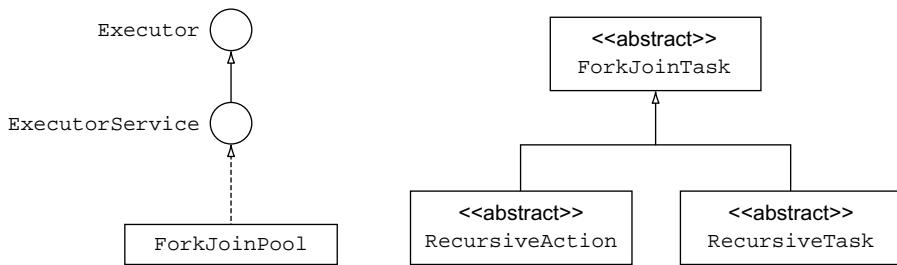


Figure 11.6 Classes implemented in the fork/join framework

a *work-stealing algorithm*. In this algorithm, when worker threads run out of tasks, they steal tasks from other worker threads to avoid blocking waiting threads.

Figure 11.5 shows how a fork/join pool maintains a common task queue but reduces contention by implementing a work-stealing algorithm. Individual threads maintain their own task queue using Deque. When a task forks a new thread, it's pushed to the thread's deque. When threads are waiting for a task to join, they pop one of their tasks, instead of simply waiting. When threads run out of all the tasks in their deque, they steal a task from the tail of another thread's deque.

Class `ForkJoinPool` provides the entry point for submissions from non-`ForkJoinTask` clients, as well as management and monitoring operations. Your problem-solving class should be a subclass of `ForkJoinTask`. `ForkJoinTask` is an abstract base class for tasks that run within a `ForkJoinPool`. A `ForkJoinTask` can be compared to a much lighter-weight version of a thread. A large number of `ForkJoinTasks` can be executed by a smaller number of actual threads in a `ForkJoinPool`. Figure 11.6 shows class `ForkJoinPool` and class `ForkJoinTask` and its two subclasses, `RecursiveTask` and `RecursiveAction`. `RecursiveAction` is used for computations that don't return a result, and `RecursiveTask` is used for computations that return a result.

Let's apply the fork/join framework to a simple example of calculating the sum of an integer array, by dividing it into smaller tasks of calculating the sum of its subarrays that execute as asynchronous tasks. In this example we'll work with `ForkJoinPool` and `RecursiveTask`.

Listing 11.2 Implement fork/join framework using `ForkJoinPool`

```

import java.util.concurrent.*;
public class CalcSum extends RecursiveTask<Integer> { ←
    private int UNIT_SIZE = 15;
    private int[] values;
    private int startPos;
    private int endPos;

    public CalcSum(int[] values) {
        this(values, 0, values.length);
    }

    protected Integer compute() { ←
        if (endPos - startPos < UNIT_SIZE) { ←
            int sum = 0;
            for (int i = startPos; i < endPos; i++) { ←
                sum += values[i];
            }
            return sum;
        } else { ←
            int mid = (startPos + endPos) / 2;
            CalcSum left = new CalcSum(values); ←
            CalcSum right = new CalcSum(values); ←
            left.invoke(); ←
            right.invoke(); ←
            return left.join() + right.join();
        }
    }
}

```

**CalcSum extends
RecursiveTask-
<Integer>.**

```

public CalcSum(int[] values, int startPos, int endPos) {
    this.values = values;
    this.startPos = startPos;
    this.endPos = endPos;
}
@Override
protected Integer compute() {
    final int currentSize = endPos - startPos;
    if (currentSize <= UNIT_SIZE) {
        return computeSum();
    }
    else {
        int center = currentSize/2;
        int leftEnd = startPos + center;
        CalcSum leftSum = new CalcSum(values, startPos, leftEnd);
        leftSum.fork(); 2 For CalcSum, overridden method compute() returns an integer value.

        int rightStart = startPos + center+1;
        CalcSum rightSum = new CalcSum(values, rightStart, endPos);
        return(rightSum.compute() + leftSum.join()); 3 leftSum.join waits until it returns a value; compute is main computation performed by task.

    }
}

Private helper method 5 private Integer computeSum() {
    int sum = 0;
    for (int i = startPos; i < endPos; i++) {
        sum += values[i];
    }
    System.out.println("Sum(" + startPos + "-" + endPos + "):" + sum);
    return sum;
}

public static void main(String[] args) {
    int[] intArray = new int[100];
    java.util.Random randomValues = new java.util.Random();

    for (int i = 0; i < intArray.length; i++) {
        intArray[i] = randomValues.nextInt(10);
    }
    ForkJoinPool pool = new ForkJoinPool(); 6 Instantiates a ForkJoinPool.

    CalcSum calculator = new CalcSum(intArray);

    System.out.println(pool.invoke(calculator)); 7 invoke() awaits and obtains result
}
}

```

Depending on how the asynchronous tasks are scheduled on a system, each execution of the preceding code might yield a different output. Here's one of the probable outputs:

```

Sum(89-100) : 56
Sum(76-88) : 46
Sum(64-75) : 62
Sum(39-50) : 48
Sum(51-63) : 43

```

```
Sum(26-38) : 32
Sum(13-25) : 57
Sum(0-12) : 54
398
```

In the preceding code, ❶ defines class `CalcSum`, which extends `RecursiveTask<Integer>`. This parameter, `Integer`, is the type of the value returned by method `compute()`. Method `compute()`, defined at ❷, is called once for the main task when `invoke()` is called on a `ForkJoinPool` instance, passing it an object of `CalcSum`. It also gets called when `fork()` is called on a `RecursiveTask` instance. Method `compute()` calculates the size of an array and compares it with a unit size (15 in this case). If it's less than this unit size, it computes the sum of its array elements using `computeSum()` ❸. If the array size is greater, it creates an instance of `CalcSum`, passing it the left half of the array, and the code at ❹ calls `fork()` on it. Calling `fork` executes a `RecursiveTask` asynchronously. Then method `compute()` creates another instance of `CalcSum`, passing it the right half of the array. At ❺, it calls `compute()` on the right part and `join()` on the left part. Calling `compute()` will recursively create (left and right) `CalcSum` objects, if it still needs to be divided into smaller tasks. Calling `join()` will return the result of the computation when it's done. Method `main()` creates an array `intArray` and initializes it by generating random integer values using class `Random`. The code at ❻ creates a `ForkJoinPool`. The code at ❼ calls `invoke()`, which executes the given task, returning its result on completion.



NOTE The preceding code example is for demonstration purposes. The fork/join framework recommends 100–10,000 computational steps in the `compute` method so that its work-stealing algorithm and `join()` can work effectively.

The order of execution of calling `compute()` and `join()` is important in the preceding code. Do you think the preceding code results in equally efficient code if you modify this execution order? Take a look in the next “Twist in the Tale” exercise.

Twist in the Tale 11.3

What is the result of replacing method `compute()` with the following modified method `compute()` from listing 11.2?

```
protected Integer compute() {
    final int currentSize = endPos - startPos;
    if (currentSize <= UNIT_SIZE) {
        return computeSum();
    }
    else {
        int center = currentSize/2;
        int leftEnd = startPos + center;
        CalcSum leftSum = new CalcSum(values, startPos, leftEnd);
        leftSum.fork();
    }
}
```

```
        int rightStart = startPos + center+1;
        CalcSum rightSum = new CalcSum(values, rightStart, endPos);
        return(leftSum.join() + rightSum.compute());
    }
}
```

- a The code might generate an incorrect sum of array elements.
 - b The code will always generate the correct sum of array elements.
 - c The code won't benefit from the fork/join framework.
 - d The code will throw a runtime exception.
-

This coverage of the fork/join framework brings an end to the concurrency topics that you need to know for the exam.

11.5 Summary

To develop thread-safe, high-performance, and scalable applications, Java's low-level threading capabilities are insufficient. In this chapter, we used collections from Java's `java.util.concurrent` package to effectively resolve common concurrency issues. We worked with multiple concurrent classes and interfaces, such as `BlockingQueue`, `ConcurrentMap`, and `ConcurrentSkipListMap`.

By relying on synchronized code blocks to coordinate access between threads, you can't develop scalable applications. You learned about explicit `Lock` objects used to wait to acquire a lock on an object's monitor. `Lock` objects also support interruptible lock waits, nonblock-structured locks, multiple condition variables, lock polling, and scalability benefits.

The `Executor` framework enables decoupling of task submission with task execution. By using this framework, you can create tasks using interfaces `Runnable` and `Callable`. These tasks are submitted to `Executor` to launch new tasks. `ExecutorService` extends `Executor` and adds methods to manage the lifecycle of tasks and executors. `ScheduledExecutorService` extends `ExecutorService` and supports future or periodic execution of tasks. `Future` represents the state of asynchronous tasks and can be used to query their status or cancel them. You use class `Executors` to call utility and factory methods for interfaces `Executor`, `ExecutorService`, and `ScheduledExecutorService`.

The fork/join framework extends the existing Java concurrency package, supporting hardware parallelism, a key feature of multicore systems. The fork/join framework isn't intended to replace or compete with the existing concurrency classes from the `java.util.concurrent` package. It works by breaking down a larger task into smaller tasks recursively, processing each unit of the task and then combining back the results.

REVIEW NOTES

Concurrent collection classes

- `BlockingQueue` defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue or retrieve from an empty queue.
- `ConcurrentMap` is a subinterface of `java.util.Map` that defines useful atomic operations.
- `ConcurrentMap` operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization.
- The standard general-purpose implementation of `ConcurrentMap` is `ConcurrentHashMap`, which is a concurrent analog of `HashMap`.
- `ConcurrentNavigableMap` is a subinterface of `ConcurrentMap` that supports approximate matches.
- The standard general-purpose implementation of `ConcurrentNavigableMap` is `ConcurrentSkipListMap`, which is a concurrent analog of `TreeMap`.
- A concurrent collection helps avoid memory consistency errors by defining a happens-before relationship between an operation that adds an object to the collection and subsequent operations that access or remove that object.

Locks

- `Lock` and `ReadWriteLock` are interfaces.
- `ReentrantLock` and `ReentrantReadWriteLock` are concrete classes.
- `Lock` objects offer multiple advantages over the implicit locking of an object's monitor. Unlike an implicit lock, a thread can use explicit lock objects to wait to acquire a lock until a time duration elapses.
- `Lock` objects also support interruptible lock waits, nonblock-structured locks, lock polling, and scalability benefits.
- Method `lock()` acquires a lock on a `Lock` object. If the lock isn't available, it waits until the lock can be acquired.
- Method `lock()` is comparable to intrinsic locks because it waits until a lock can be acquired on a `Lock` object.
- Call `unlock` on a `Lock` object to release its lock when you no longer need it.
- If you don't call `unlock()` on a `Lock` object after acquiring a lock on it, the code will still compile successfully.
- Method `tryLock()` tries to acquire a lock on a `Lock` object, and returns immediately a boolean value specifying whether it could obtain the lock or not.
- Watch out for the use of methods `acquire()`, `acquireLock()`, `release()`, and `releaseLock()` on the exam. None of these is a valid method. Because the terms `acquire` and `release` are used to discuss methods `lock()`, `unlock()`, `tryLock()`, and `lockInterruptibly()`, these terms might be used on the exam to confuse you.

- Methods `lockInterruptibly()` and `tryLock(long time, TimeUnit unit)` of the `Lock` interface enable you to specify a waiting timeout or to try and acquire a lock while being available for interruption.
- Extrinsic locks or the lock on `Lock` objects can be acquired across methods.
- An interface, `ReadWriteLock` maintains a pair of associated locks, one for read-only operations and another for write-only operations.
- The `ReadWriteLock` interface doesn't extend `Lock` or any other interface.
- The `ReadWriteLock` interface defines only two methods, `readLock()` and `writeLock()`.
- A `ReentrantReadWriteLock` has a read and a write lock associated with it. You can access these locks (reference variables of type `Lock`) by calling its methods `readLock()` and `writeLock()`.
- You can acquire read locks until a write lock has been acquired on a `ReadWriteLock` object.
- `WriteLock` is an exclusive lock; it can be acquired by only one thread when no read thread has been acquired.
- The `java.util.concurrent` package defines multiple classes that support atomic operations of read-compare/modify-write on single variables.
- Other commonly used classes defined in the `java.util.concurrent.atomic` package are `AtomicInteger`, `AtomicBoolean`, `AtomicLong`, `AtomicIntegerArray`, `AtomicLongArray`, and `AtomicReference<V>`.
- The `java.util.concurrent.atomic` package doesn't define classes by the names `AtomicShort`, `AtomicByte`, `AtomicFloat`, or `AtomicDouble`. These invalid class names might be used on the exam.

Executors

- The `Executor` framework enables decoupling of task submission with task execution.
- By using this framework, you can create tasks using interfaces `Runnable` and `Callable`.
- The `Runnable` interface defines method `run()` and the `Callable` interface defines method `call()`.
- `Executor` allows you to decouple task submission and its execution.
- The `Executor` interface defines only one method, `void execute(Runnable)`.
- You can define your own execution policy by implementing `Executor`'s method `execute()`.
- Comparing the `Runnable` and `Callable` interface, method `run()` of `Runnable` doesn't return a value and can't throw a checked exception. But method `call()` of `Callable` can return a value and throw a checked exception.
- If you don't want `Callable` to return a value, you can create it using `Callable<Void>`, defining the return type of `call()` as `Void` and returning `null` from it.

- The `ExecutorService` interface extends the `Executor` interface and defines methods to manage progress and termination of tasks that are submitted to it.
- Implemented using `ExecutorService`, thread pools use a pool of worker threads to execute new tasks. If the pool runs out of worker threads, the submitted task waits for a worker thread to become available.
- Thread pools prevent spawning of new threads to execute each new submitted task, thereby avoiding overwhelming the scheduler.
- Class `Executors` defines utility and factory methods for interfaces `Executor`, `ExecutorService`, and `ScheduledExecutorService`.
- `ScheduledExecutorService` extends `ExecutorService` and supports future or periodic execution of tasks.
- Future represents the state of an asynchronous task and can be used to query its status or cancel it.
- `ScheduledExecutorService` can schedule `Callable` or `Runnable` tasks that execute just once, after a given delay.
- `ScheduledExecutorService` can schedule only `Runnable` tasks (not `Callable` tasks) that can execute multiple times, starting its first execution after an initial delay and subsequent execution after the specified period.

Parallel fork/join framework

- The parallel fork/join framework makes the best use of multicore processors.
- The fork/join framework extends the existing Java concurrency package, supporting hardware parallelism, a key feature of multicore systems.
- The fork/join framework isn't intended to replace or compete with the existing concurrency classes from the `java.util.concurrent` package.
- The fork/join framework works by breaking down a larger task into smaller tasks recursively, processing each unit of the task, and then combining back the results.
- You can define a recursive task using either `RecursiveTask` or `RecursiveAction`. `RecursiveAction` is used to define tasks that don't return a value. `RecursiveTask` returns a value.
- You can instantiate a fork/join pool by using `ForkJoinPool` and passing it the number of processors it should use. By default, `ForkJoinPool` uses all the available processors.
- Execution of `RecursiveAction` or `RecursiveTask` starts when you call `invoke` on `ForkJoinPool`, passing it a `RecursiveAction` or `RecursiveTask` object, which calls their `compute()`.
- Method `compute()` determines whether the task is small enough to be executed or if it needs to be divided into multiple tasks. If the task needs to be split, a new `RecursiveAction` or `RecursiveTask` object is created, calling `fork` on it. Calling `join` on these tasks returns their result.

SAMPLE EXAM QUESTIONS

Q 11-1. Which of the following options is a thread-safe variant of ArrayList?

- a ConcurrentList
- b ConcurrentArrayList
- c CopyOnReadWriteArrayList
- d CopyOnReadArrayList
- e CopyOnWriteArrayList

Q 11-2. Which line of code, when inserted at //INSERT CODE HERE, will ensure that on execution, single or multiple MyColors instances won't throw a ConcurrentModificationException at runtime?

```
class MyColors extends Thread{  
    static private Map<Integer, String> map;  
    MyColors() {  
        //INSERT CODE HERE  
        map.put(1, "red");  
        map.put(2, "blue");  
        map.put(3, "yellow");  
    }  
    public void iterate() {  
        Iterator iter = map.keySet().iterator();  
        while(iter.hasNext()) {  
            Integer key = (Integer)iter.next();  
            String val = (String)map.get(key);  
            System.out.println(key + "-" + val);  
            add(4, "green");  
        }  
    }  
    public void add(Integer i, String v){  
        map.put(i, v);  
    }  
    public void run() {  
        iterate();  
    }  
}
```

- a map = new HashMap<Integer, String>();
- b map = new NoExceptionHashMap<Integer, String>();
- c map = new ConcurrentHashMap<Integer, String>();
- d map = new ConcurrentHashMap<Integer, String>();
- e map = new CopyOnWriteHashMap<Integer, String>();
- f None of the above

Q 11-3. Which of the following concurrent collection classes can you use to implement the producer-consumer design pattern?

- a WaitNotifyQueue
- b BlockingQueue
- c LinkedBlockingQueue
- d ArrayBlockingQueue
- e ConcurrentBlockingQueue
- f ProducerConsumerQueue
- g ProducerConsumerList

Q 11-4. Examine the following code and select the correct options.

```
1. class UseConcurrentHashMap {
2.     static final ConcurrentHashMap<Integer, String> map =
3.             new ConcurrentHashMap<>();
4.     static {
5.         //code to populate map
6.     }
7.     static void manipulateMap(Integer key, String value) {
8.         if (!map.containsKey(key))
9.             map.put(key, value);
10.    }
11. }
```

- a Because code at lines 2 and 3 uses ConcurrentHashMap, operations in method manipulateMap() are thread safe.
- b Operations in manipulateMap() will be thread safe if lines 8 and 9 are replaced with map.putIfAbsent(key, value);.
- c If line 2 is replaced with the following code, there won't be any concurrency issues with manipulateMap(): static final ConcurrentHashMap<Integer, String> map =.
- d Removing code at lines 4, 5, and 6 will ensure calling manipulateMap() won't override any values.

Q 11-5. Which of the following methods from class AtomicLong can be used to atomically compare values and modify them?

- a compareAndSet
- b compareAndModify
- c modifyAndCompare
- d compareValuesAndSet
- e compareAndSetModifySafely

Q 11-6. Examine the following code and select the correct answer options.

```
class University {
    Lock myLock = new ReentrantLock(); //1
    static List<String> students = new ArrayList<>();
    public void add(String newStudent) {
        myLock.lock(); //2
        try {
            students.add(newStudent);
        }
        finally {
            myLock.unlock(); //3
        }
    }
}
```

- a At line 1, if a reference variable lock is assigned an object of ReentrantReadWriteLock, it won't make a difference.
- b The code at line 2 tries to acquire a lock on object myLock. If the lock isn't available, it waits until the lock can be acquired.
- c The code at line 2 tries to acquire a lock on object myLock. If the lock isn't available, it returns immediately.
- d If the code at line 3 is removed, the code will fail to compile.

Q 11-7. Select the incorrect statements.

- a The ReadWriteLock interface maintains a pair of associated locks, one for read-only operations and another for write-only operations.
- b A read-only lock may be held simultaneously by multiple reader threads, irrespective of whether there are any writing processes in progress or not.
- c A write-only lock is an exclusive lock. It can be acquired by only one thread.
- d The ReadWriteLock interface defines only two methods, `readLock()` and `writeLock()`.

Q 11-8. Objects of which of the following can be executed by an Executor?

- a class Order implements Runnable {
 public void run() {}
}
- b class Order implements Callable {
 public void call() {}
}
- c class Order implements Executable {
 public void execute() {}
}
- d class Order implements Schedulable {
 public void schedule() {}
}

Q 11-9. Which of the following can be used to execute a Runnable or Callable object after an initial delay of 10 minutes?

- a ForkJoinPool
- b ScheduledExecutorService
- c TimedExecutorService
- d SchedulableExecutorService

Q 11-10. Code in which of the following options can be inserted at //INSERT CODE HERE?

```
class MyService {  
    public static void main(String args[]) {  
        ExecutorService service = Executors.newFixedThreadPool(5);  
        service.submit(new Task());  
    }  
}  
//INSERT CODE HERE  
  
a class Task implements Executor {public void execute() {} }  
b class Task implements Executable {public void execute() {} }  
c class Task implements Submittable {public void submit() {} }  
d class Task implements Callable<Void> {public Void call() {return null;} }  
e class Task implements Callable<Integer> {public Integer call() {return 1;} }  
f class Task implements Callable {public void call() {} }
```

Q 11-11. Which of the following objects can be passed to ForkJoinPool.invoke()?

- a ForkJoin
- b ForkJoinThread
- c ForkJoinTask
- d AbstractForkJoinTask
- e RecursiveThread
- f RecursiveAction
- g RecursiveTask

Q 11-12. Which of the following problems make a good candidate to be solved using the fork/join framework?

- a Problems that can be divided into independent tasks
- b Problems that work with a lot of external files
- c Problems the subtasks of which rely on synchronization to calculate their results
- d Problems that need to connect with an external server to choose their courses of action

ANSWERS TO SAMPLE EXAM QUESTIONS

A 11-1. e

[11.1] Use collections from the `java.util.concurrent` package with a focus on the advantages over and differences from the traditional `java.util` collections

Explanation: Options a, b, c, and d are incorrect because the Java API doesn't define these interfaces or classes.

A 11-2. d

[11.1] Use collections from the `java.util.concurrent` package with a focus on the advantages over and differences from the traditional `java.util` collections

Explanation: Option (a) is incorrect because iterators returned by `HashMap` are fail-fast. If a collection changes (addition, deletion, or modification of elements) after you retrieved an iterator, it will throw a `ConcurrentModificationException`.

Options (b) and (e) are incorrect because these use invalid class names.

Option (c) is incorrect because `ConcurrentMap` is an interface; therefore, it can't be instantiated.

Option (d) is correct because iterators returned by `ConcurrentHashMap` are fail-safe. If a collection changes (elements are added, deleted, or modified) after the iterator is created, they don't throw any runtime exception. These iterators work with a clone of the collection rather than working with the collection itself.

A 11-3. c, d

[11.1] Use collections from the `java.util.concurrent` package with a focus on the advantages over and differences from the traditional `java.util` collections

Explanation: Methods `put()` and `take()` of the `BlockingQueue` interface specify in-built locking support to be provided by all the implementing classes to implement the producer-consumer design pattern. `LinkedBlockingQueue` and `ArrayBlockingQueue` are concrete implementations of the `BlockingQueue` interface.

A 11-4. b

[11.1] Use collections from the `java.util.concurrent` package with a focus on the advantages over and differences from the traditional `java.util` collections

Explanation: Option (a) is incorrect. The individual operations of classes `ConcurrentHashMap`, `containsKey`, and `put` are atomic and safe to be used concurrently by multiple threads. But when these methods are used together in `manipulateMap`, it doesn't guarantee thread safety for the time between these two method calls. Multiple calls to

atomic operations don't form an atomic operation. You need external locking or synchronization to make such operations thread safe.

Option (b) is correct because method `putIfAbsent()` in `ConcurrentHashMap` combines two steps: checking for the existence of a key and replacing its value in a method, and synchronizing the key-value pair access across threads.

Options (c) and (d) are incorrect because these steps will not make a difference in this case.

A 11-5. a

[11.2] Use Lock, ReadWriteLock, and ReentrantLock classes in the `java.util.concurrent.locks` package to support lock-free, thread-safe programming on single variables

Explanation: Options (b), (c), (d), and (e) are incorrect because they're invalid method names.

A 11-6. b

[11.2] Use Lock, ReadWriteLock, and ReentrantLock classes in the `java.util.concurrent.locks` package to support lock-free, thread-safe programming on single variables

Explanation: Option (a) is incorrect because, on assigning an object of `ReentrantReadWriteLock` to reference variable `myLock`, the code won't compile. Class `ReentrantReadWriteLock` and the `Lock` interface are unrelated. `ReentrantReadWriteLock` doesn't implement `Lock`, directly or indirectly.

Option (d) is incorrect because you should unlock an object when you no longer require a lock on it. If you don't, your code will compile anyway.

A 11-7. b

[11.2] Use Lock, ReadWriteLock, and ReentrantLock classes in the `java.util.concurrent.locks` package to support lock-free, thread-safe programming on single variables

Explanation: Option (b) is incorrect because the read-only lock may be held simultaneously by multiple reader threads, as long as no write lock is acquired.

A 11-8. a

[11.3] Use Executor, ExecutorService, Executors, Callable, and Future to execute tasks using thread pools

Explanation: Option (b) is incorrect because the `Executor` interface defines just one method, `void execute(Runnable)`. It only accepts objects of `Runnable`, not `Callable`.

Options (c) and (d) are incorrect because they use invalid interface names.

A 11-9. b**[11.3] Use Executor, ExecutorService, Executors, Callable, and Future to execute tasks using thread pools**

Explanation: Option (a) is incorrect because ForkJoinPool doesn't support execution of a scheduled delay of tasks.

Options (c) and (d) are incorrect because they use invalid class names.

A 11-10. d, e**[11.3] Use Executor, ExecutorService, Executors, Callable, and Future to execute tasks using thread pools**

Explanation: You can pass either Runnable or Callable to method submit() of ExecutorService. Options (d) and (e) correctly implement the Callable<T> interface. Callable's method call() returns a value. If you don't want it to return a value, implement Callable<Void>, define the return type of call() to Void, and return null from it.

A 11-11. c, f, g**[11.4] Use the parallel fork/join framework**

Explanation: ForkJoinPool.invoke() accepts objects of class ForkJoinTask, which is an abstract class. RecursiveAction and RecursiveTask are abstract subclasses of ForkJoinTask.

A 11-12. a**[11.4] Use the parallel fork/join framework**

Explanation: A fork/join framework is best suited to be applied to problems that can be divided into independent tasks that can be executed in parallel. Subtasks that need to work a lot with I/O or network connections can't utilize the processors optimally because they might be blocked waiting for an I/O operation. Similarly, when subtasks need to synchronize among themselves, they defeat the whole purpose of executing in parallel to speed up the execution.